Enhancing Cloud Security with Trusted Execution

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Stefan Brenner
geboren am **9.4.1987**
in **Blaubeuren**

Eingereicht am: **22.09.2020**
Disputation am: **14.12.2020**
1. Referent: **Prof. Dr.-Ing. Rüdiger Kapitza**
2. Referent: **Dr. Valerio Schiavoni**

(2020)

Contents

Kurzfassung

Die steigende Popularität von Cloud Computing führt zu immer mehr Nachfrage und auch strengeren Anforderungen an die Sicherheit in der Cloud. Nur wenn trotz der technischen Möglichkeiten eines Cloud Anbieters über seine eigene Infrastruktur ein entsprechendes Maß an Sicherheit garantiert werden kann, können Cloud Kunden sensible Daten einer Cloud Umgebung anvertrauen und diese dort verarbeiten. Das vorherrschende Paradigma bezüglich Sicherheit erfordert aktuell jedoch zumeist, dass der Kunde dem Cloud Provider, dessen Infrastruktur sowie den damit verbundenen Softwarekomponenten komplett vertraut. Während diese Vorgehensweise für manche Anwendungsfälle einen gangbaren Weg darstellen mag, ist dies bei Weitem nicht für alle Cloud Kunden eine Option, was nicht zuletzt auch die Annahme von Cloud Angeboten durch potentielle Kunden verlangsamt.

In dieser Dissertation wird nun die Anwendbarkeit verschiedener Technologien für vertrauenswürdige Ausführung zur Verbesserung der Sicherheit in der Cloud untersucht, da solche Technologien in letzter Zeit auch in preiswerteren Hardwarekomponenten immer verbreiteter und verfügbarer werden. Es ist jedoch keine triviale Aufgabe existierende Anwendungen zur portieren, sodass diese von solch gearteten Technologien profitieren können, insbesondere wenn neben Sicherheit auch Effizienz und Performanz der Anwendung berücksichtigt werden soll. Stattdessen müssen Anwendungen sorgfältig unter verschiedenen spezifischen Gesichtspunkten der jeweiligen Technologie umgestaltet werden. Aus diesem Grund umfasst diese Dissertation zunächst eine Diskussion verschiedener Sicherheitsziele für Cloud-basierte Anwendungen und eine Übersicht über die Thematik „Cloud Sicherheit“. Zunächst wird dann das Potential der ARM TrustZone Technologie zur Absicherung einer Cloud Plattform für generische Anwendungen untersucht. Anschließend wird beschrieben wie eigenständige und bestehende Anwendungen mittels vertrauenswürdiger Ausführung am Beispiel SGX abgesichert werden können. Dabei wurde der Fokus auf relevante Metriken gesetzt, die die Sicherheit und Performanz einer solchen Anwendung beeinflussen. Zuletzt wird, ebenfalls basierend auf SGX, eine vertrauenswürdige „Serverless“ Cloud Plattform vorgestellt und damit auf aktuelle Trends für Cloud Plattformen eingegangen.

Abstract

The increasing popularity of cloud computing also leads to a growing demand for security guarantees in cloud settings. Cloud customers want to be able to execute sensitive data processing in clouds only if a certain level of security can be guaranteed to them despite the unlimited power of the cloud provider over her infrastructure. However, security models for cloud computing mostly require the customers to trust the provider, its infrastructure and software stack completely. While this may be viable to some, it is by far not to all customers, and in turn reduces the speed of cloud adoption.

In this thesis, the applicability of trusted execution technology to increase security in a cloud scenario is elaborated, as these technologies are recently becoming widespread available even in commodity hardware. However, applications should not naively be ported completely for usage of trusted execution technology as this would affect the resulting performance and security negatively. Instead they should be carefully crafted with specific characteristics of the used trusted execution technology in mind. Therefore, this thesis first comprises the discussion of various security goals of cloud-based applications and an overview of cloud security. Furthermore, it is investigated how the ARM TrustZone technology can be used to increase security of a cloud platform for generic applications. Next, securing standalone applications using trusted execution is described at the example of SGX, focussing on relevant metrics that influence security as well as performance of such an application. Also based on SGX, in this thesis a design of a trusted serverless cloud platform is proposed, reflecting the latest evolution of cloud-based applications.

1 Introduction

Undoubtedly, the interest, popularity, adoption and prevalence of all various kinds of cloud computing [79] has been constantly increasing in the recent years [87, 25, 46] due to the benefits for all involved parties: On the one hand, customers gain a high level of flexibility, cost-efficiency, a large feature set and easy access to many different kinds of computing resources. On the other hand, the providers are able to increase the average load on single machines by collocation of multiple customer's applications on the same machine to achieve better efficiency by reduction of idle times on the machines. Furthermore, the principle of "economy of scale" allows the providers to invest efficiently in computing resources and relay pricing benefits to their customers [25].

The most important cornerstone technology that enabled the earliest form of cloud computing was the virtualisation technology. It introduces the concept of virtual machines that multiplex single physical machines for usage with multiple (competitive) parties, and led to Infrastructure-as-a-Service (IaaS) platforms such as the open source project *OpenStack* [104]. These solutions demand for a general set of requirements such as resource control and isolation, equivalence of execution compared to physical machines and a high amount of native instructions for efficiency [93]. The introduction of virtualisation enabled higher average machine load, and thus, more efficient resource usage without frequently wasting CPU time to idle tasks or Input/Output (I/O) waiting. Cloud providers could buy server hardware at large scale and benefit from bulk discounts, at the same time, cloud customers are safeguarded from the risk of large upfront hardware investment and gain flexible scalability at low risk as cloud resources are usually billed on a pay-as-you-go basis.

The evolution of the cloud computing paradigm also brought Platform-as-a-Service (PaaS) that comprises a platform on top of which users can deploy their applications without the need to maintain the server's hardware components and Operating System (OS). The most modern variant of cloud computing is Function-as-a-Service (FaaS) that even more than previous approaches reduces the maintenance efforts required by cloud customers and offloads them to the cloud provider. In this paradigm, the customers write their applications in the form of multiple small functions that are deployed and executed on a cloud platform without realising the server boundaries as

the platform automatically scales the functions across machines depending on current load and does not require merely any resources when the function is unused—hence also called *serverless computing*.

1.1 Security and Trust in Cloud Computing

The aforementioned benefits of the various different kinds of cloud computing aside, security is one of the most crucial factors in cloud computing and also significantly influences cloud adoption by customers and their trust in the technology [87, 55, 49, 82, 64]. In general, the problem with such security issues is that they affect and risk the confidentiality of the data that is processed in the cloud. This may lead to unwanted leakage of data to entities that were not supposed to have access to that data and loss of control over where the data resides or propagates to. Security issues like that can have various different reasons as described in the following.

Software bugs in the cloud software stack are only one possible source of such a security issue and comprise not only the source code of the customer's application but also source code of all other involved software components installed and managed by the cloud provider. For example, the cloud provider usually maintains the system's firmware, the OS, the Virtual Machine Monitor (VMM) and there are components responsible for managing the cloud platform, and monitoring and accounting tasks.

Furthermore, the cloud provider's personnel itself might be a security risk as they could access privileged software components on the machines, and thus, could also access the memory contents of all processes. The cloud provider's personnel might even have physical access to the machines which paves the way for physical attacks such as the *cold boot attack* [33]. Such kinds of privileged or physical accesses to data of cloud customers might not only be initiated by the cloud provider, but could also be originating from higher level such as governmental authorities that request access to that data[1]—in some cases even forcing the cloud provider not to notify their customers.

Finally, large commercial clouds with global influence own several data centres all over the world and customers often have no technical means to control where exactly the data resides and propagates to. An interest of cloud providers in geographically distributed backups and mirroring of data is comprehensible, however, customers can only trust in contracts that set policies for data propagation.

Currently the dominating security model in clouds is to settle constraints and requirements in the form of contracts. The customer then has to trust the provider [55]

[1] `https://transparencyreport.google.com/user-data/overview`

to satisfy those contracts as violations would lead to financial penalties. However, this approach may not always be an option: for example, data privacy in medical applications is priceless, as the processed patient data has a high significance to the affected patient for their whole life time and beyond. Another example would be usage of cloud services by governmental authorities that would also have no interest in financial compensation if sensitive data such as intelligence data or political strategies would be leaked to hostile entities. In addition, even if financial penalties are negotiated, data leakage might have defamatory effects, as end users might loose trust into a cloud application even if the application provider is not responsible for the leak—thereby trust by clients into an application being a priceless asset. Therefore, in order to allow such sensitive data processing in a cloud environment and thereby protecting the involved data from unauthorised accesses, a secure and trusted cloud environment is required, and security and trust must be enforced by technical means.

1.2 Enabling Trusted Execution in the Cloud

In order to protect against the various attack vectors described above, this thesis investigates the design criteria of a secure cloud platform by leveraging trusted execution technology. For the context of this thesis, the general properties of a secure cloud platform in this spirit is briefly described in the following. At the same time featuring cloud data processing according to those criteria constitutes the central goal of this thesis.

One of the most important and also the most basic security goal of a secure cloud platform is the protection of the integrity of the software running in the cloud. This allows a customer to be assured that the software claimed to run in the cloud by the provider is actually what the customer expects to run and not in any way modified by anyone. Without this property other security goals such as confidentiality are meaningless because it is not guaranteed that an encryption is actually executed. A secure and trusted cloud platform also aims at protecting the confidentiality of the processed data from unauthorised accesses by the cloud provider according to the above described attack vectors such as privileged software and physical attacks directly to the machines as well as software bugs in all involved components. Transitively, this also affects data access requests originating from governmental authorities forcing the cloud provider to hand out the data. Furthermore, such a secure and trusted cloud requires a procedure for cloud customers to establish trust into the execution environment in the cloud by technical means. Hence, a way to attest the execution environment in the cloud must be provided to the customers, in order for them to be able to verify the environment prior to the deployment of applications and sensitive data.

In order to achieve those goals, for example, a Trusted Platform Module (TPM) [114] could be leveraged to execute a secure boot of the hardware platform that allows the customer to verify the whole software stack in the cloud. For this to work, the cloud provider had to install a TPM from a trusted (third) party on all machines in question and publish the source code of all involved software components. Then, the customer could verify the source code and decide whether to trust it or not, before requesting a cryptographic proof that this source code is actually deployed on the cloud platform and nothing else. Only after such a successful attestation the customer would deploy the software and sensitive data on that cloud platform.

However, the above described approach of a fully trusted cloud software stack poses several disadvantages to all involved parties and is practically infeasible to achieve as it required the provider to publish all source code that might even contain business secrets of the provider that the customer's are not supposed to know. Therefore, it is unrealistic to assume that any cloud provider would be willing to publish the complete source code of all deployed software components on her cloud machines. Furthermore, this attestation procedure had to be repeated for each update to any of the software components. This is posing a high effort to both the provider and the customers particularly because software updates have to be assumed to be required regularly and quite frequently to fix new security vulnerabilities. In addition, due to the high complexity of a full cloud software stack it would be rather infeasible for the customer to gain a significant understanding of whether the cloud protects sensitive data sufficiently or not. Finally, the cloud software stack would constitute a very large amount of source code that has to be trusted. This comprises software components that are used to manage the cloud infrastructure, as well as accounting and monitoring of the cloud, and constitutes a very large Trusted Computing Base (TCB), which is a security risk in general, as more lines of code usually correlate with a higher probability of exploitable security vulnerabilities [63, 108, 107, 75, 10]. Similarly, a cloud customer might not even want to deploy complete applications in a trusted cloud environment as this increases the TCB unnecessarily as well, instead only security-critical software components are desirable to be deployed and executed in a secured execution environment.

In addition to all above issues and disadvantages, a fully-trusted cloud software stack could at most only protect against unauthorised accesses by privileged software components, while it still constitutes no protection against physical attacks. However, for many use cases it is not acceptable to confide in the provider, neither with the code nor the processed data of the application, therefore this approach is not sufficient to protect the sensitive data according to our defined goals.

A more modern way to achieve the above defined security goals by creation of a trust-

worthy execution environment is provided by trusted execution technology. There are several different technologies available that allow the creation of a so called Trusted Execution Environment (TEE). Thereby, such a TEE is "trusted" in the sense that the cloud customer or user could request a proof about the integrity of the execution environment itself and the code executed within it.

In this context, a TEE is a specially secured environment for execution of trusted code components where particular hardware mechanisms are applied to, in order to increase the security of that environment. Early versions of such technologies, such as Intel TXT and ARM TrustZone, offer strong hardware-enforced isolation guarantees. However, more modern approaches such as Intel Software Guard Extensions (SGX) or AMD Secure Encrypted Virtualization (SEV) additionally provide transparent hardware-based memory encryption, raising the security to a new level. Using hardware capable of this in cloud systems, would allow a cloud provider to provide a TEE to their cloud customers. The customers could then establish trust in such a cloud TEE and deploy their software inside it, which constitutes a basic required building block of all trusted cloud environments.

1.3 Practical Usability of Trusted Execution in Cloud Computing

The sole availability of trusted execution technology is not enough for creating secure software or platforms. Instead, the individual expectations and assumptions of all involved parties must be respected. For example, the cloud provider will not comply in publishing all of its software stack, neither will she accept to install arbitrary software on her machines. On the other hand, customers of the cloud provider want a proof of the trustworthiness of the platform in order to ensure that their deployed software and the potentially sensitive processed data is protected. In addition, a reasonable usage of the technology and respecting its specific properties is essential in order to design a system that fulfils the above defined security properties while providing good performance.

Just to mention an example, ARM TrustZone only provides a single trusted environment that can not trivially be used for multiple software artefacts from competitive cloud customers. As multi tenancy is a crucial requirement for cloud platforms in order to achieve high average load and efficient resource usage, TrustZone would be only usable in a cloud environment if the secure world can be multiplexed to run multiple competing software components in parallel and isolated from each other. In addition it must be ensured that those software components can be attested individually from each other by their respective owner.

Another example is the fact that Intel SGX enabled transparent memory encryption

only for a quite limited range of up to 256 MB of the system's memory. Once this range is exceeded a very costly paging process is required that evicts pages to regular system memory and poses a huge performance penalty to the affected applications. This also shows that the software design has to be tailored for usage in a cloud setting to make use of the available hardware capabilities in an efficient way.

1.4 Enhancing Cloud Security with Trusted Execution

As motivated by the above discussion and problem description, this thesis targets the investigation of security measures with assistance of various trusted execution technologies to allow processing of sensitive data in an untrusted cloud environment. As stated, this is done under the main assumption that the cloud provider is not trusted by the customers. Also we assume that customers do not mutually trust each other. This requires data protection in a way that the provider has never access to the plain text data, neither by exploiting its privileged control over software components running in the cloud, nor by physically attacking or tampering with the server machines in question. Inherently this also covers access requests by higher level authorities trying to force the cloud provider to release sensitive customer data. At the same time we aim at minimising the overall TCB while providing sensitive data processing securely in an untrusted cloud as the reduction of the TCB also minimises the attack surface and probability of critical software bugs.

The primary security goal is to protect the confidentiality of (possibly) sensitive data that is processed in the cloud. This constitutes a much more difficult to achieve requirement than only storing sensitive data in a cloud environment, as it is not enough to encrypt the data on the client side before transmission, instead the cloud must be able to execute computations on the sensitive data.

In order to ensure that the confidentiality of processed data is protected, the integrity of the cloud platform and its software components must be guaranteed and attestable by remote customers. If the integrity of the cloud software platform could not be guaranteed, the cloud provider could install additional software components or alter existing ones in a way that violates data confidentiality and leaks data to unauthorised parties.

The main problem statement for this thesis is divided into the following research challenges: Firstly, it is investigated how trusted execution technology can be used in a cloud setting, at the example of the ARM TrustZone technology. Then, porting challenges for applications to run with increased security by assistance of trusted execution technology are investigated for several existing applications both with ARM TrustZone and the new Intel SGX technology. Finally, the combination of trusted execution with

modern software architectures is investigated at the example the FaaS paradigm. These research challenges are described in more detail in the following sections.

1.4.1 Research Challenge 1: Enabling usage of trusted execution technology in an untrusted cloud environment

In order to enable usage of trusted execution in an untrusted cloud setting, the envisioned target properties of such a trusted cloud platform have to be defined. This comprises a general security discussion of attack vectors relevant to cloud computing platforms and defines the target security goals of our secure cloud platform. For this purpose we developed the TrApps platform based on the ARM TrustZone technology, that uses this technology to allow execution of general purpose secure components tied to regular cloud applications running in an insecure environment. The TrApps platform is described in more detail in Chapter 3 and has been published in the form of the following research paper:

- *TrApps: Secure Compartments in the Evil Cloud* in the proceedings of the Workshop on Security and Dependability of Multi-Domain Infrastructures (XDOM0'17).

1.4.2 Research Challenge 2: Application of trusted execution technology to protect the confidentiality of sensitive data in existing cloud applications

In Chapter 4 of this thesis the application of trusted execution technology to protect confidentiality of sensitive data in existing cloud-related applications is investigated. This covers retrofitting trusted execution into existing applications with the goal of deployment on a platform similar to the previously introduced TrApps platform. Therefore it is investigated how sensitive application logic is identified and extracted from the application's code base and offloaded to a trusted environment embedded into the application. Furthermore, this chapter names and discusses various relevant properties that affect performance and security of such applications. In this scope also the novel Intel SGX technology is investigated, that opposed to ARM TrustZone also features the transparent memory encryption for increased security. The contents of this chapter have been published in the form of the following research papers:

- *Running ZooKeeper Coordination Services in Untrusted Clouds* on the 10th Workshop on Hot Topics in System Dependability (HotDep'14).

- *SecureKeeper: Confidential ZooKeeper using Intel SGX* in the proceedings of the 17th ACM Middleware conference (MIDDLEWARE'16).

- *Trusted Execution, and the Impact of Security on Performance* on the 3rd Workshop on System Software for Trusted Execution (SysTEX'18).

1.4.3 Research Challenge 3: Merging serverless cloud computing and trusted execution in an untrusted cloud context

In this thesis, modern and upcoming software architectures for cloud applications are investigated at the example of serverless cloud computing. Hereby, the goal was to analyse their suitability for usage together with trusted execution to protect the confidentiality of sensitive data. In Section 5 of this thesis, trusted execution is applied to cloud functions following the FaaS application programming paradigm. The contents of this chapter have been published in the form of the following research paper:

- *Trust More, Serverless* in the proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19).

1.4.4 Vision: New Opportunities with Trusted Cloud Platforms

The vision of this thesis is to achieve higher security in public clouds without trusting the cloud provider by incorporation of trusted execution technology into existing and new cloud applications. By this the main goal is to allow data processing of sensitive data in the cloud environment, thereby protecting that sensitive data from being accessed by the cloud provider and other unauthorised entities. This is opposed to other existing approaches that enable secure data storage in cloud environments without the ability to do data processing in the cloud, but instead download the data to trusted machines and process the data there. The goal is to enable usage of public clouds for currently impossible or impractical use cases such as medical or financial applications or governmental usage involving highly sensitive data.

2 Background

This chapter describes the relevant background for this thesis. This comprises an overview of what cloud computing is and in what forms it currently exists. In addition to that, in this chapter trusted execution technologies are introduced in general, but with a focus on the technologies used in this work by providing details only about ARM TrustZone and Intel SGX.

2.1 Cloud Computing

The general concept of cloud computing intends to deliver computing resources to cloud customers via standard networks while increasing the overall resource usage efficiency and offloading maintenance tasks to the provider [79]. Thereby, the increased resource usage efficiency is achieved by co-location of multiple customers on the same hardware, increasing the average load on the machines and preventing large idle times were the machines are unused or only on light load. At the same time a cloud provider can benefit from the *economy of scale*, as she can invest in larger quantities of hardware. Furthermore, resource provisioning to a cloud customer does not require human intervention and allows applications to scale—even automatically—to the dynamic load on the system [79]. As a result, cloud customers can focus more on the application logic of their services instead of the computing infrastructure and its maintenance. Scalability is another important benefit of cloud computing, as applications can dynamically and quickly adjust to the current load situation using seemingly infinite resources. This prevents customers from the inherent risk of upfront investments in hardware [25] that might only be needed for short periods of very high load, and instead provides them a flexible *pay-as-you-go* payment model.

2.1.1 Cloud Computing Models

Initially, a precondition for cloud computing was the availability of virtualisation technology [125] as this allows the co-location of multiple virtual machines on a single physical machine. Especially, cloud computing was made possible by *efficient* virtualisation,

that allows strong isolation of multiple competing applications from each other, and provides virtual machines that appear (almost) indistinguishable from physical machines [93]. Based on such virtualisation technology, initial cloud platforms implemented the IaaS paradigm which provides virtual machines that behave like physical machines to the customers. Customers of IaaS still manage the OS and all software running on top of it, but are relieved from the burden of infrastructure maintenance tasks, such as providing uninterruptible power supply, redundant network connection and proper cooling. In contrast to commercial IaaS offerings, Eucalyptus [83] and later OpenStack [104] opened the paradigm to the open source community and researchers.

In order to reduce the management overhead of the customers in IaaS clouds, PaaS as the next step in the evolution of cloud computing, provides a platform to the customer where she can deploy and run complete applications [39]. Thereby, the customer is not involved in the management of the platform underneath, which is fully managed by the cloud provider instead. This includes installation, configuration and management of the complete software stack comprising the OS and all software running on top of the OS. PaaS clouds also promote the recent shift from monolithic software designs towards modular architectures such as microservice and FaaS architectures. The former has the goal of splitting large applications into smaller (micro)services that can work independently from each other [32], the latter splits applications into single standalone functions that are deployed in the FaaS cloud platform [13].

2.2 Trusted Execution Technology

Processor manufacturers have implemented various trusted execution technologies in the past years, such as the ARM TrustZone technology [9] or Intel TXT [1]. More recently Intel released the SGX technology [78] that features transparent encryption of TEE memory. Lately, AMD has introduced the Secure Memory Encryption (SME) [2] technology supporting similar memory encryption but without integrity protection.

Trusted execution generally aims at supplying a trusted environment for execution of sensitive code isolated from the otherwise untrusted execution environment. In general terms such a trusted environment created with trusted execution technology is called a Trusted Execution Environment (TEE) [110] and intended to run alongside a rich OS providing security services to it. Security services of an application component running

[1] `http://www.intel.de/content/www/de/de/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html`, last accessed 06/2020.

[2] `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, last accessed 06/2020

inside a TEE are called trusted application and are supposed to be consumed by an untrusted application component running in the regular and untrusted environment called client application.

A TEE can be trusted only after verification of the code that is running inside it. This requires to proof to a (remote) entity—the *attester*—what code is actually executed inside the TEE along with a proof of the integrity of the TEE's platform. It must be guaranteed that the trusted execution technology itself is in place and working correctly—especially that it is *not* emulated. Also, it must be proved to the attester that only the intended code is loaded into the TEE and this code has not been altered in any way. For this purpose a hash of affected software components can be created by *measuring* that code and presenting the cryptographically signed hash as a proof to the attester [72].

On a system that supports the creation of TEEs, the components that need to be trusted are summarised under the term Trusted Computing Base (TCB). This primarily comprises all source code running in the TEE including for example Software Development Kit (SDK) libraries, but also involves the hardware platform providing the ability to create a TEE. In this thesis, this term also denotes a measure of the amount of code that runs inside a TEE.

When being externalised, confidentiality of the processed sensitive data inside a TEE can be protected by encrypting that data, for example with a special encryption key only known to the trusted component. This key should be persisted by storing it outside the trusted environment in order to survive TEE restarts. However, before storing such sensitive data outside the TEE, measures to protect that data have to be taken. For this purpose TEE technologies usually support a feature called *data sealing*. Data sealing denotes a special kind of encryption tied to the hardware's identity that executes the operation. For example, data could be encrypted with a special key crafted from a hash over TEE-specific components such as the hardware and software identity and system configuration for example. This implies that the same key can only be retrieved from that very system in that very system configuration. Hence, sealed data is tied indivisibly to the TEE and can only be decrypted in that controlled and trusted environment.

In the following sections, the two trusted execution technologies used for the prototypes as part of this thesis are described in more detail. On ARM platforms, the TrustZone technology [9] (described in Section 2.3) allows isolated execution of trusted code inside a TEE which is called *secure world*. For x86 systems the Intel Software Guard Extensions (SGX) technology [78] (described in Section 2.4) provides the notion of *secure enclaves* as isolated compartments inside a user process.

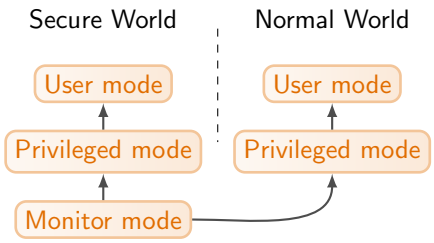


Figure 2.1: TrustZone System Architecture.

2.3 ARM TrustZone

TrustZone [9] is a hardware security extension for ARM processors that splits the processor virtually into the two worlds, the *secure world*—that constitutes the TEE—and the *normal world*. As can be seen in Figure 2.1, both worlds have their own user and privileged execution mode for execution of system and user software components. During execution, the CPU switches between the two worlds via the so called *monitor* mode, which is responsible for a controlled world switch without leakage of any secrets across the world boundary. The TrustZone technology features hardware-based isolation between those two worlds, which affects the main memory, system bus, peripheral devices, and the interrupt configuration.

After powering up the board and the usual firmware initialisation routines, the board enters secure world and first load a secure world kernel. Once secure world has finished its initialisation procedure, a world switch to normal world is executed and the boot process continues initialising the normal world kernel.

TrustZone allows splitting the system's main memory and assigning an arbitrary amount of memory to secure world. Any memory access attempts from normal world to secure world memory are prevented by the architecture. However, secure world is higher privileged and allowed to access not only its own memory but also the memory that is assigned to normal world. This enables communication between the two worlds of TrustZone via a shared memory range located in the normal world memory region. As the secure and normal world CPU contexts maintain individual caches, memory shared between worlds must be handled in a CPU cache-agnostic way.

Usage of TrustZone requires splitting the software stack into two distinct components that run in a time-multiplexed manner on the same CPU that switches between these two worlds regularly. There exist explicit world switches that directly call the monitor mode from the control flow of the software using the `smc` instruction. Once `smc` is called, the CPU switches to monitor mode, which is implemented by the system architect and responsible for saving and restoring the CPU context of normal and

secure world, respectively.

In addition to explicit world switches, the TrustZone Interrupt Controller (TZIC) can be used to configure interrupts and assign them to either of the two worlds. By this, interrupts cause an implicit asynchronous world switch upon arrival and enforce interrupt handling in the respective world as configured in the TZIC. For example, a timer interrupt can be assigned to secure world, to ensure liveness of secure world, while Network Interface Controller (NIC) interrupts may be assigned to the normal world.

The TrustZone technology allows protecting against many hack attacks [9] (e.g. malware in normal world). Isolation of secure and normal world is implemented by time slicing on CPU level and a security-aware system bus which is able to assign peripherals to one of the two worlds [72]. This improves security by allowing the offloading of sensitive application logic to a TEE, however, TrustZone only supports one single TEE. There is only one secure world and virtualisation is not possible within secure world but only in normal world for some newer boards.

Trusted boot on ARM platforms (c.f. [9]) can be achieved by signing all relevant binary images and verifying them during the boot process. This is usually done using asymmetric cryptography and a vendor's public key stored on the hardware platform in a way that prevents replacement by an attacker. By this, it can be ensured that the boot process is executed as intended and with valid images only, based on a so called "root of trust" on the hardware. A root of trust, for example, could be implemented as a public key of the software vendor integrated into the hardware platform or burned into one-time-programmable memory [9].

2.4 Intel SGX

On the x86 architecture, the instruction set extension Intel Software Guard Extensions (SGX) [78, 28] allows the creation of multiple TEEs called *secure enclaves* embedded into user space processes. SGX works on a special memory region called Enclave Page Cache (EPC) which is reserved during boot time for storage of the memory pages of all secure enclaves on the platform. The Memory Encryption Engine (MEE), integrated into each SGX-capable CPU applies transparent memory encryption with integrity protection to this special memory range [45]. By this, SGX removes the memory from the TCB and reduces it to only the executed code and the CPU [76].

SGX introduces the two CPU instructions "`encls`" for kernel space operations and "`enclu`" for user space operations, each with a number of leaf functions identified by the set value of the `RAX` CPU register. Table 2.1 and Table 2.2 shows the most important SGX instructions and data structures, respectively.

Leaf function	Description
ECREATE	Create enclave
EADD	Add memory page to enclave
EEXTEND	Measure 256 bytes of memory
EINIT	Finalise enclave creation process
EREMOVE	Remove memory page from enclave
EENTER	Enter enclave
EEXIT	Leave enclave
ERESUME	Resume execution of enclave

Table 2.1: Overview of important SGX instructions.

Data structure	Description
SGX Enclave Control Store (SECS)	Enclave meta data
Thread Control Structure (TCS)	Thread meta data
State Save Area (SSA)	Enclave thread context and state
SIGSTRUCT	Enclave signature structure

Table 2.2: Overview of important SGX data structures.

2.4.1 Life Cycle and Interaction of Enclaves

A secure enclave's *life cycle* [28] begins with a call to ECREATE which declares the size of the enclave and its base address inside the user space application's address space, which is also called Enclave logical range (ELRANGE). Meta information about the enclave to be created is stored in a data structure called SECS which is provided to ECREATE as an argument. Afterwards, memory pages containing the code of the enclave and also empty memory pages are added to the created enclave using the EADD instruction. Thereby, EADD works similar to memcpy as it copies pages from regular memory into the enclave's memory inside the EPC. The EINIT instruction completes the enclave creation process and prevents further pages from being added to the enclave. However, only after calling the EINIT instruction threads are allowed to enter the enclave. This restriction applies especially to SGX v1 and is partly relaxed in SGX v2 as described later.

SGX enclaves are supposed to be seen as secure libraries containing a number of functions that are executed securely in the protected environment of an enclave. In order to call a function inside an SGX enclave, a so called *entry point* must be used, which is described during the enclave creation process and enforces controlled enclave entries. Essentially, an enclave entry point describes a memory offset within the enclave's memory range that can be jumped at from the outside. However, enclave entries are only pos-

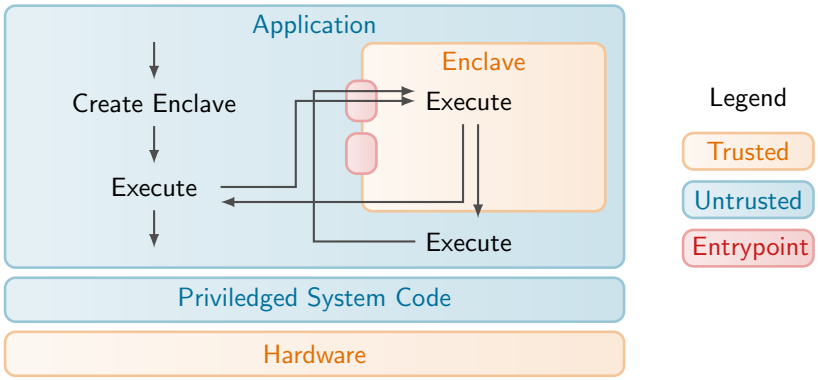


Figure 2.2: SGX enclave life cycle and interaction.

sible using the `EENTER` instruction along with a pointer to a data structure called `TCS` which describes thread meta data and depicts one of the enclave's entry points to be called. After finishing processing inside an enclave, the control flow leaves the enclave using the `EEXIT` instruction, which jumps back to the untrusted environment where the enclave has been entered beforehand. During `EENTER` and `EEXIT` it is the developers responsibility to protect sensitive data leakage via CPU registers. Therefore, sensitive register contents should be cleared before leaving the enclave. As entering and exiting an enclave requires a Translation Lookaside Buffer (TLB) flush in order to prevent sensitive data leakage, crossing the enclave boundary implies a delay of several thousand CPU cycles [121]. Figure 2.2 illustrates the interaction of a user space application with an embedded SGX enclave.

When an interrupt is triggered on a CPU that is currently in enclave mode, a so called Asynchronous Exit (AEX) is performed. An AEX interrupts enclave execution and stores the enclave's current execution state inside a SSA data structure that is referenced in the TCS data structure. This comprises for example the current instruction pointer and all CPU registers, such that enclave execution can later resume at the point of interruption. The CPU registers are also automatically overwritten by the CPU with a synthetic and clean state before the enclave is exited, such that no sensitive data leaks to the outside. After exiting the enclave, the interrupt is handled normally and the control flow returns to the so called Asynchronous Exit Pointer (AEP) once the interrupt handling is completed. The AEP then is responsible for resuming enclave execution at the earlier interrupted state by using the `ERESUME` instruction and referencing the respective TCS data structure. `ERESUME` enters the enclave and restores the state at the moment of interruption and eventually continues enclave execution at the point of interruption.

SGX also allows multi threading inside secure enclaves, i.e. multiple threads entering

the same enclave simultaneously. This requires that enough TCS data structures have been added to the enclave during its creation process. A TCS data structure must be "available" in order to be used by a thread to enter an enclave, i.e. it can only be used by one thread at a time. The TCS data structure is responsible for storing meta data about the thread and also denotes the entry point into the enclave to be used. Furthermore, the TCS data structure references a list of SSA data structures for storage of thread execution contexts and state in case of asynchronous enclave exits. As those enclave exits can even occur nested, a list of such SSA data structures is required.

2.4.2 Memory Protection and Management

The memory region that backs all memory pages of all enclaves of a platform is called EPC and resides in the Processor Reserved Memory (PRM): a special region of regular system memory reserved during boot time by the system's firmware and communicated to the OS kernel. The responsibility of the untrusted OS is to manage this memory and also to maintain all the page tables even though it can not access the plain text contents of that memory. This range of memory is currently architecturally limited to a maximum of 256 MB which is mostly usable for enclaves with a small fraction of remaining space reserved for security meta data [45].

The enclave memory in the EPC is protected by the MEE using keys randomly generated after CPU start-up. The MEE protects the confidentiality and also the integrity of the enclave pages by usage of cryptographic operations. This also comprises protection against replay attacks to enclave memory.

An SGX-capable CPU applies a set of measures to the memory pages of enclaves inside the EPC in order to protect their confidentiality and integrity. This comprises the prevention of direct jumps of control flow across the enclave border, to and from the untrusted environment. However, SGX permits read and write access from within an enclave to untrusted memory of the host process the enclave lives in. Whenever a page of an enclave in the EPC is accessed by an untrusted source, so called *abort page semantics* apply: read operations return `0xFF`, while write operations are ignored.

SGX allows eviction of enclave pages from the EPC to regular system memory by the OS when the available EPC is exceeded. This *EPC paging* requires the re-encryption of all evicted pages, in order to protect the confidentiality and integrity of those pages in untrusted memory as well. Re-encryption is required as the CPU can not access the encrypted cipher text of the data but only sees the transparently encrypted plain text of the data, hence, an additional encryption of the data must be applied before storing the enclave data in untrusted memory [78]. The process of EPC paging is also replay

protected by a *Version Array (VA) page*, that is created during the eviction of a page from the EPC and allows swapping the page back into EPC only once. Due to the required re-encryption of all pages during EPC paging, this procedure poses a high performance impact on SGX applications experiencing it.

2.4.3 SGX System Support and Enclave Development

Along with SGX-capable CPUs, Intel published the open source Intel SGX SDK [53] that supports developers during the enclave development process. The Intel SGX SDK introduces the terms *ecall* and *ocall*, for enclave calls from the untrusted application into the enclave and outside calls from the enclave to the outside untrusted environment, respectively. In order to support the developer during the enclave development, the Intel SGX SDK generates trusted and untrusted helper code for ecalls and ocalls such that they resemble an Remote Procedure Call (RPC)-like calling model. This generated code is derived from a Domain-specific Language (DSL) that describes the enclave's interface with the untrusted environment—the Enclave Description Language (edl).

Since SGX enclaves can only execute in user space (ring 3), it is not possible to execute system calls directly from an enclave. Instead, the control flow has to leave the enclave and execute the call from the untrusted user space and forward the OS's result back inside the enclave. Therefore, the Intel SGX SDK provides a modified `libc` library that runs inside an enclave and bridges the gap to the outside OS via ocalls.

The Intel SGX instruction set extension was released in version 1 with the Intel Skylake CPU generation. Already at that time, the SGX specification described some additional functionality available in SGX version 2. This comprises additional SGX leaf functions to support dynamic memory management of SGX enclaves. With SGX version 2 it is possible to add and remove memory pages to an already initialised enclave (c.f. `EINIT` in Section 2.4.1). Added pages could also be TCS pages, allowing SGX version 2 applications to dynamically add more threads during enclave runtime. Furthermore, SGX version 2 allows changing the access permissions of enclave pages which SGX version 1 allows only once at the time when a memory page is added to the enclave.

2.4.4 Integrity and Attestation

During enclave creation (c.f. Section 2.4.1), memory pages added to the enclave using `EADD` are supposed to be *measured* using the `EEXTEND` instruction [78]. This instruction *extends* the value of the *enclave identity* by calculation of a checksum of a 256 Byte chunk of the memory page's content concatenated with the previous value. The enclave identity value describes the enclave's memory contents and the sequence of SGX in-

structions that were called during enclave creation and is held in the MRENCLAVE field of the enclave's SECS data structure. This value is finalised by the EINIT instruction that completes the enclave creation process and can not be changed afterwards.

In order for the EINIT instruction to accept an enclave and initialise it correctly, it expects a valid EINITTOKEN [28], which is issued by a special enclave provided by Intel called the *launch enclave* and comprises meta information about the enclave to be initialised such as the expected MRENCLAVE value. Only on provision of a valid EINITTOKEN the enclave is marked initialised by the architecture and ready to be entered by user space threads. EINIT also expects a signature by the enclave developer's private key which is delivered in the form of the SIGSTRUCT data structure [28, 6]. During EINIT, the signature inside SIGSTRUCT is verified and MRENCLAVE is compared to the signed MRENCLAVE inside the SIGSTRUCT. During the EINIT process, the MRSIGNER value is written into the SECS which denotes the identity of the enclave certificate's signer [6, 28].

Attestation allows to establish trust into an SGX enclave by proving the enclave's integrity and the integrity of the underlying SGX hardware to the attester. The attestation process can be executed between two enclaves on a platform (*local attestation*) or between an enclave and a remote entity (*remote attestation*). Though, on SGX platforms, remote attestation is based internally on local attestation.

Local attestation proves an enclave's MRENCLAVE value to another enclave on the same platform. For this the attesting enclave—attester—sends its own MRENCLAVE value to the enclave to be attested. Upon reception, this enclave creates a *report* using the EREPORT instruction, which is destined and distinct for the attester's MRENCLAVE and sends it back to the attester. This report proves that the hardware is trustworthy and comprises the enclave's identities and attributes, up to 256 Bit arbitrary user data and a Message Authentication Code (MAC). The attester now retrieves its *report key* using the EGETKEY instruction which allows the verification of that MAC. This proves that the attested enclave is running on the same platform. Afterwards, the attester compares the MRENCLAVE value included in the report with the expected value and is then assured of the integrity of the attested enclave. Finally, this process can be repeated in reverse direction to establish a mutually trusted relationship between the two enclaves.

The process of remote attestation of SGX enclaves is based on the local attestation described in the previous paragraph. However, as a first step of remote attestation, the enclave to be remotely attested is first locally attested by a special enclave provided by Intel—the *Quoting Enclave*. This enclave can create a so called *quote* after successful local attestation, that constitutes a cryptographic proof of the identity of the attested enclave that can be verified by a remote party. Upon reception of such a quote, the remote attester can verify it using the Intel Attestation Service (IAS) provided by Intel and

compare the MRENCLAVE value included in the quote with the expected value. Thereby, remote attestation is inherently based on trust in Intel as the processor manufacturer, that equips every SGX-capable CPU with an individual key during manufacturing and is also responsible for the IAS. However, it is also possible to establish a third party attestation service without run-time dependencies to Intel [5, 101].

Since an enclave's data in the EPC is backed by (volatile) DRAM, it is essential to be able to persist data outside the enclave. However, confidentiality and integrity of an enclave's assets stored outside it must still be protected. For this purpose, Intel SGX offers special keys supposed to encrypt data persisted outside the enclave's boundary in a process called *sealing* [6]. Those keys can be accessed by an enclave using the EGETKEY instruction, and differentiate between the enclave and the sealing identity. Sealing to the enclave identity only permits access to instances of the very same enclave as the key is derived from the enclave's identity (MRENCLAVE). In contrast, sealing to the sealing identity permits access also to other enclaves with the same sealing authority as the key is derived from MRSIGNER. With the help of those keys, the enclave can persist data securely outside its reach by using any cryptographic operation of its own choosing.

In general, SGX was tailored to protect the confidentiality and integrity of sensitive user data in enclaves. Using SGX allows to assume a very strong attacker model, that even covers physical access to the machine and allows the OS itself to be considered malicious. However, as SGX enclaves are executed within regular user processes on top of a commodity OS, they are helpless against availability attacks such as a denial of service attacks as the underlying OS can simply decide to not schedule an enclave. In addition to that, side channel attacks are excluded from SGX's threat model [28], leaving defence mechanisms against such kinds of attacks to the enclave developers.

3 Trusted Execution in the Cloud

Cloud computing [79] has gained traction and popularity in the recent years [87, 25, 46]. This is due to its benefits for all involved parties, as the cloud providers benefit from efficient resource usage and the economy of scale, while the customers can offload many maintenance tasks to the cloud provider and focus more on their main targets. Thereby, cloud computing allows the customers to flexibly rent all kinds of computing resources they need, when they need them and only as long as they actually need them. However, security is still a crucial requirement and as it can not be guaranteed in a cloud environment yet [86], this influences cloud adoption significantly and even renders some use cases impractical or infeasible in public clouds [87, 55, 49, 82, 97, 46, 62].

The demand for a secure and trusted cloud platform is high and could enable new cloud usage scenarios that are impossible at the moment. This requires the cloud providers to establish a platform that can guarantee certain security properties to their customers. Therefore, it is not enough to only define constraints in contracts that both, the provider and the customer sign, instead it is required to enforce security properties such as confidentiality of data by technical means. While confidentiality-protected data storage in an untrusted cloud is feasible relatively easily by client-side data encryption before uploading the data to the cloud, it is a much harder challenge to allow data processing of sensitive data in such an untrusted environment.

In this thesis, a trusted cloud platform is approached that allows sensitive data processing as described above without trusting the cloud provider. This means the cloud provider should be restricted by technical means from accessing the customer's data. This not only prevents the cloud provider from accessing the data despite not being allowed to, but also removes the burden of being able to do so off the cloud provider. The latter is particularly relevant in case of higher level access requests by a governmental authority that pursues mass surveillance, for example.

If such a trusted cloud platform existed, it would enable new opportunities for usage scenarios that are not possible right now. For example, currently if sensitive data is supposed to be processed in a public cloud environment it has to be gauged if data leakage can be accepted and compensated through financial penalties. This could be a relevant option for cloud usage by industry that experience profit collapses on data

leakage. However, it might not be a viable option for medical applications that involve patient data that are relevant to a patient and even her descendants for decades. Another example could be governmental cloud usage, that, for example, could incorporate intelligence data. In both these cases, impaired entities of data leakage would probably not be interested in financial compensation and suffer from irreplaceable damage. Currently, since there are no technical means to protect data from such risks, such cloud usage is not possible.

One straight forward approach to achieve a trusted cloud platform would be the attestation of the whole software stack. However, for several reasons this is particularly impractical in a cloud scenario: the cloud provider would be forced to publish the source code of all his management and infrastructure software including the hypervisor to allow this. Though this is problematic since the competition between cloud providers results in highly customised software [3] which might also be considered a business secret. For example Amazon used to rely on a highly customised version of the Xen hypervisor [14] and just recently switched to a likewise customised KVM-based [59] hypervisor. In addition, any updates to attested software would require remote attestation again. Furthermore, the attested software stack would be huge, resulting in a higher probability for exploitable security vulnerabilities [75]. Thereby, this applies to software components running on commercial cloud machines that are responsible for accounting and monitoring of the cloud platform. Due to these disadvantages this approach to a trusted cloud is to be considered impractical.

Instead in this thesis we try to establish a trusted cloud platform by usage of trusted execution technology in an otherwise untrusted cloud environment. On top of our envisioned trusted cloud platform we intend to run partitioned applications with only a minimal trusted component in order to reduce the overall TCB and achieve high security. In addition to that, for the same reasons, we also put emphasis on minimising the TCB of the trusted cloud platform itself.

In this chapter we first discuss various origins of security issues in a cloud setting, and define the properties of a trusted cloud platform. Then, we present our trusted cloud platform *TrApps*, which implements the defined security properties and allows the execution of partitioned cloud applications with trusted components.

3.1 Cloud Security Issues

There are several possible sources of security risks in a cloud setting that go beyond a regular software's security risks. Some are caused by the co-location of multiple software artefacts on the same hardware platform and various kinds of interactions be-

tween them. Others are partly caused by the benefits that cloud computing offers to its customers, such as the inherent safety through redundancy of running a software component in multiple data centres, that causes a loss of control over where data resides globally. In the following we describe different sources of cloud security risks.

3.1.1 Risk of Software Vulnerabilities

Just as with traditionally deployed software, software deployed in a cloud is at risk through software bugs that could be exploited to gain unauthorised access to sensitive data. As studies have shown, the risk of exploitable security vulnerabilities grows with the complexity and amount of code to be trusted and is inevitable [88, 63, 108, 107, 75, 10]. This applies to any software, whether it is deployed traditionally or in the cloud. However, a cloud deployment of software artefacts adds new sources of such security risks, as other software components that are not in the reach of the customer can contain bugs as well and jeopardise the data privacy of the customer's data.

There are many different aspects of software components out of the customer's reach. Firstly, there are software components that the cloud provider requires in order to offer her services to customers, such as the system firmware, the OS, device drivers and the hypervisor used for virtualisation. In addition to that, there are software components the cloud provider deploys on her platform in order to be able to maintain the platform itself. For example, a cloud platform consists of many services and software components, that are responsible for storing data, controlling virtual machines and managing network traffic. Also, the cloud provider needs software components to monitor the platform, control virtual machine placement decisions, enforce resource isolation and fairness of resource usage, as well as accounting in order to bill the customer exactly for what resources she actually used.

All these listed software components may contain bugs that risk the data privacy of some or all data in the cloud. Furthermore, the large amount of features and possibilities in a cloud also lead to a very large code base of the software involved to deliver those capabilities to customers in a flexible way. Therefore, a cloud platform consists of a large number of software components of different origin and adds up to a huge code base that could contain exploitable software vulnerabilities that endanger data privacy.

3.1.2 Human-induced Attack Vectors

Another factor of security that is certainly more specific to cloud environments compared with traditional software deployments are attack vectors caused by additional individuals that have access to the cloud platform. Firstly, this comprises the cloud

provider's personnel which is required to have access to the platform as they are responsible for maintaining correct operation of the cloud. Cloud providers will limit this to a preferably small amount of persons but there will usually be at least some that are permitted to enter the data centres and have privileged access to the machines if not even physical access. Especially physical access is the hardest to defend against as it allows for example cold boot or evil maid attacks [43, 33]. Furthermore, there is no guarantee to customers about who else may or may not have any kind of access to the cloud platform granted by the provider, such as cleaning personnel or fire-fighters, for example. After the *Snowden* revelations, it is also known that intelligence agencies could have online access to cloud providers [71]. More broadly, also governmental authorities might obtain access to cloud data due to law enforcement and control over the cloud provider's legal sphere. Hereby, a cloud provider might be forced by governmental authorities to grant access to customer's data without even telling the customer about those conditions. This area of security risks is particularly critical across country borders and could only be regulated by contracts but is practically infeasible to audit by a customer. Hence, cloud customers currently always simply have to trust their provider.

In addition to that, there are also security risks caused by hackers that try to penetrate the cloud systems and applications deployed in the cloud either remotely from the outside via the internet, or even with help from inside the cloud provider. This is in general not quite different from any other internet connected service running on private infrastructure, except for the fact that deployment in the cloud, as described above, adds more software components to the overall TCB and implies more individuals with access to system components, increasing the probability of exploitable vulnerabilities.

3.1.3 Cloud-specific Attack Vectors

In addition to the above mentioned security risks originating from software bugs and human individuals, there are some factors specific to cloud deployments. For example, the basic idea of cloud computing comprises the offloading of management tasks to the cloud provider. However, this inherently also leads to a lack of knowledge of the cloud customer about where exactly the machines are located and where the data is stored. Clearly, commercial cloud providers such as Amazon offer the customers to choose from a set of data centres, but again it is required that the customer trusts that the cloud provider stores the data only in the chosen data centre. The customer can never know for sure if the provider executes geographical replication of data across data centres, countries or even continents [8, 55] or if the provider abides an agreement with the customer not to do such replication. Admittedly, it can be beneficial to do geographical

replication as it improves the end user's latency, fault tolerance and disaster recovery [1], however, it might be also problematic if data crosses country's borders by this and enters distinct legal areas.

It is also not trivial to gain or keep control over data and achieve *assured deletion* [95] in a cloud environment. Backup and data mirroring strategies by the cloud provider might—as mentioned—involve copying that data across country borders and continents, and across legal boundaries as well. Also, there is usually neither any guarantee, that data is actually deleted and also deleted from all old backup versions when a cloud customer deletes any data from the cloud, nor is this even feasible especially for versioned backups made by the cloud provider. Due to several layers of redundancy and data replication in the cloud in order to increase availability and fault tolerance as well as the overall performance, deleting data gets an increasingly complex task.

In general, for a cloud customer it is crucial to keep control over where data propagates to and who has access to that data. In some cases, contracts can be concluded to arrange penalties upon violation of the customer's demands, or illegitimate data dissemination is prohibited by law (e.g. General Data Protection Regulation (GDPR)). However, there are cases where penalties are not enough and a violation of the customer's constraints is unacceptable. For example, usage of cloud services by governmental authorities processing sensitive data in the cloud, or medical applications that involve patient data, might not be able to tolerate any data leakage at all. As of now, in such cases the usage of cloud services for data processing is not possible.

3.2 Basic Properties and Assumptions of a Trusted Cloud Platform

The above discussion of possible security risks in current commercial clouds motivates the search for a trusted cloud environment that could allow sensitive data processing leveraging the manifold benefits of cloud computing and bypassing those risks. In this section, the essential properties of such a trusted cloud platform are discussed and the assumptions and basic parameters are defined.

3.2.1 Terminology and Assumptions

For this thesis, integrity of the execution environment is considered a crucial requirement for a trusted cloud platform. This comprises the need for a genuine hardware platform that correctly implements its instruction set and especially instruction set extensions for trusted execution, such as ARM TrustZone and Intel SGX. Therefore, it is required to trust the hardware manufacturers that they implement the hardware cor-

rectly and include no back doors or other technical means appropriate to divert data to unauthorised recipients. It is beyond the scope of this thesis to discuss how hardware components could be verified to work correctly, hence, it is assumed here that the hardware manufacturers are trustworthy, as only then, a trusted environment for execution of security-sensitive software components can be established.

In the scope of this thesis, it is assumed that the implementation of the hardware, and the CPU in particular, is bug-free and not compromised in any way. This is not self-evident as recent research as proven [68, 60, 117], so it is assumed that the CPU works as intended and known security flaws are eliminated by, for example, microcode updates. Furthermore, side channel attacks [126] that leak sensitive data via unintended covert and unauthorised channels are not considered in this work.

The involved entities of a secure cloud scenario comprise several different trust domains. Firstly, there are the *hardware manufacturers* that offer hardware components that are purchased by the cloud providers. The *Cloud Provider* installs the hardware in her data centres and deploys cloud management software on those machines. Next, the *Cloud Service Provider* purchases the right to use computing resources of the cloud provider for her own use. Finally, the *Cloud User* accesses services deployed in the cloud. Note, that a Cloud User could be a Cloud Service Provider as well but does not necessarily have to be. Essentially any of the named entities have to trust the hardware manufacturer as flaws in the hardware design affect the integrity of all execution and isolation measures important for all parties.

Throughout this thesis, a *trusted cloud platform* denotes a cloud platform (component) that is able to process sensitive data and protects that data from unauthorised accesses in an otherwise untrusted cloud environment. Unauthorised accesses, in this context, denote any accesses to the plain text of the sensitive data by entities not supposed to access it, such as the cloud provider, other customers of the cloud provider with software executed on the same machines or hackers accessing the cloud platform via the internet or internally in the form of insider attacks.

In this context, a *cloud application* is an application deployed and maintained by a cloud customer or cloud service provider in a public cloud environment. The cloud application comprises only the actual application logic that processes the data as intended by the cloud customer, and does not include the software components required by the cloud provider to maintain the availability and functionality of the cloud platform itself. Also, the TCB of the cloud application shall not comprise most of the cloud provider's software components in order to allow keeping the TCB as lean as possible and excluding unnecessary components.

3.2.2 Attacker Model and Trust Relationships

Many attacker models for cloud computing assume that the preliminary risk for cloud applications originates from external entities attacking the cloud via the network. This work allows a stronger adversary model that not only includes hackers, but also assumes that the cloud provider is not trusted by its customers. This means that the provider shall not have any opportunity to access the plain text data that is uploaded by the customer and processed in the cloud at any time. Even though this might in some rare cases be acceptable, cloud customers most frequently do not want the provider to be able to access their data. Indeed cloud providers in many cases even do not have an incentive to be able to access customer's data, as this prevents them from plausibly denying access requests originating from higher level authorities such as governments. Naturally, a cloud provider could always destroy data or shut down offered services at any time, which would affect the availability of the services in question and constitute a breach of contract with the customer that would lead to financial compensations, but this shall not affect the data privacy in any way. Furthermore, since cloud customers do not trust other customers of the same provider, those shall not be able to access sensitive data as well. It is only required to trust the cloud provider to meet the availability and accounting properties negotiated in contracts between the provider and the customer.

It is essential that sensitive data is processed inside a TEE in order to comply with the above defined data privacy and confidentiality requirements. Only if the platform and software that processes the sensitive data can be trusted, confidentiality of data can be maintained. However, also the integrity of the software that is executed inside a TEE is crucial for the confidentiality requirements, as incorrect or malicious software could leak sensitive data from a trusted environment to the outside world. Hence, a TEE should be entrusted with sensitive data only after successful verification of the integrity of the software running inside it. Remote attestation is key in establishing trust into a remote trusted execution environment, by measuring the code running inside it and proving the result to a remote party.

3.2.3 Security Goals

Cloud applications as described above should be in large parts untrusted with only small trusted components that are deployed in a TEE. This reduces the overall TCB of the cloud application which constitutes a significant security factor. This is partly due to the much easier verification of a small TCB as it simply comprises less that is to be investigated. Furthermore, a leaner TCB will lead to less security-critical vulnerabilities as less code statistically contains less bugs (c.f. Section 3.1.1). Also, bugs in the

untrusted code base can not lead to data leakage or unauthorised accesses as those code components are not supposed to be able to access the plain text but only the cipher text of the sensitive data. Finally, even a formal verification of code components would only be feasible for a smaller TCB, simply because otherwise this procedure would be impractical from a performance point of view. The details of designing software to run in an untrusted cloud environment with a small TCB are discussed in Chapter 4 while this chapter focusses on designing an ARM TrustZone-based cloud platform.

3.2.4 Cloud-specific Aspects

The trusted cloud platform shall be able to keep the defined security goals while executing the workloads of multiple competitive cloud customers on the same hardware platform, as it is essential for cloud computing to co-locate multiple workflows on the same physical machines in order maximise the overall efficiency. This requires strong isolation of applications from each other by the cloud platform, particularly the ones owned by distinct customers, such that customers can not access other customer's data.

It is assumed that cloud customers deploy their applications remotely in the cloud by uploading applications in binary form, and control runtime parameters using the cloud platform's designated Application Programming Interfaces (APIs). Furthermore, clients are supposed to access the deployed applications via the internet or external network, by issuing requests to the cloud applications that are then processed accordingly.

For practical usage of such a trusted cloud platform it is also significant to achieve a reasonable performance as experienced by the end users. Therefore, client's requests shall be processed and responded to in reasonable time with preferably low latency and high overall throughput. Clearly, due to data encryption in the trusted cloud, a significant performance penalty is to be expected, however, the secure cloud's performance shall at least be in the same order of magnitude as insecure clouds. For example, there are also homomorphic encryption schemes that allow computation directly on encrypted data without knowing the plain text, even arbitrary operations are possible with the latest approaches [38]. However, usually these suffer from impractically low performance, and thus, are not (yet) suitable for realistic scenarios [64, 73].

3.3 Usage of ARM TrustZone in the Cloud with TrApps

In this thesis, trusted execution technologies like ARM TrustZone or Intel SGX are used as a basic building block to establish TEEs for the deployment of trusted components. Those technologies are widespread available in commodity hardware, as op-

posed to special-purpose trusted components such as the ones used in TrustedDB [12] or CheapBFT [57]. With the availability of server-grade ARM systems [50] that feature high performance while operating energy- and cost-efficiently [89], ARM architectures are no longer mobile-only platforms but pose a competitor to the established and dominating x86 server platforms. For example, *Scaleway* is the first vendor to provide x86-64, ARMv7 as well as ARMv8 bare-metal and virtual cloud servers[1]. Also, the aspect of diversity alone motivates the investigation of the ARM architecture next to x86, however, ARM also features the benefit of being an open platform with chip manufacturers from diverse origins. Therefore, this section investigates how the ARM TrustZone technology can be used in an untrusted cloud setting in order to secure sensitive applications.

With TrustZone, ARM designed a trusted execution technology that splits the execution into the two worlds, secure world and normal world. This allows offloading parts of the software stack to the higher privileged secure world that is isolated by means of hardware from normal world. In a cloud setting, however, the requirement for multi tenancy support demands for the possibility to execute multiple secure components simultaneously in secure world. Therefore, a way to multiplex usage of secure world has to be found that allows efficient execution of secure components from distinct cloud customers in secure world at the same time.

In this section we introduce the *TrApps* platform that constitutes a trusted cloud platform using the TrustZone technology and allows the above mentioned multiplexing of the secure world in order to support the cloud-inherent multi tenancy. The TrApps platform is supposed to be maintained by the cloud provider and allows the simultaneous execution of multiple secure components provided by cloud service providers in TrustZone's secure world. Thereby, the general idea is that applications are divided into a secure and insecure part, with the secure part containing the sensitive application logic of the application. The secure part is managed by its associated insecure component which controls the secure part's life cycle in secure world.

It is the responsibility of the cloud provider to maintain availability of the TrApps platform for being used by cloud service providers and their respective users. Therefore, the cloud provider will install redundant power supply and network connectivity as well as cooling for the machines in question. Furthermore, the cloud provider has to deploy and initialise the TrApps platform on suitable machines and ensure the platform's availability to its users. For this purpose, the cloud provider will also monitor the involved systems and repair or replace them in case of failures. Finally, the cloud provider will keep track of the involved software components and test and deploy up-

[1] `https://www.scaleway.com/en/virtual-instances/arm-instances/`, last accessed 11/2019

dates in a timely manner in order to ensure that security-critical bugs are fixed.

Secure applications running on the TrApps platform comprise of secure and insecure application logic compartments. It is the general idea that a piece of application logic supposed to be executed on TrApps is investigated by its owner and sensitive parts of its application logic are identified. Then, the sensitive parts are split from the insensitive parts and installed in a separate "trusted" application component, while the remaining application logic is consolidated into an untrusted component. The two parts are supposed to be deployed in secure and non secure world of TrustZone respectively, and need to be connected with each other in order to be able to interact across the world boundary of TrustZone using an API provided by TrApps. In the context of this thesis, a trusted component inside TrApps is called *Trustlet* while the untrusted application component is referred to as *NApp*. Together, a Trustlet and NApp are called a *TrApps Application*. The main architecture of TrApps is illustrated in Figure 3.1 and described in more detail in the following sections of this chapter.

The idea is that a Trustlet is tailored to its connected NApp and augments the application logic of its NApp with sensitive application logic parts. Thereby the Trustlet is embedded into the NApp's control flow similar to a library. The NApp is also responsible for managing the life cycle of a Trustlet starting with its creation, its usage and finally its termination. A Trustlet is not directly exposed to the network, instead the NApp should accept connections and manage connection handling to a certain extent, while forwarding sensitive data from incoming requests arriving as cipher text to the Trustlet which is the only entity able to decrypt the data for processing. Hence, a secure cloud application running on TrApps is the aggregate of a secure and insecure component—a Trustlet and a NApp—that interact with each other and run in secure and normal world of TrustZone respectively. For security reasons as described above, a Trustlet is intended to be relatively small compared to its NApp.

In order to deploy and execute an application on TrApps, the application has to be partitioned or designed as a partitioned application from scratch. This application partitioning process is discussed in Chapter 4 of this thesis in depth, while this chapter covers the TrApps platform as a basic building block underneath the partitioned application on ARM architectures with TrustZone. In the remainder of this section, firstly the Genode OS framework that TrApps is based on is described. Then, the design and system architecture of TrApps is presented, followed by TrApps' cross-world communication method. Afterwards, the life cycle and programming model of TrApps applications, and a discussion about security assumptions and bootstrapping trust in the TrApps platform is given. The section is finished with work related to TrApps.

3.3.1 TrApps and the Genode OS Framework

The Genode OS framework [37] allows building secure special-purpose operating systems based on a micro kernel architecture. While initially supporting the L4 family of micro kernels, recent releases include Genode's independent micro kernel *base-hw*.

Genode provides strong process isolation by imposing a strict organisational structure on processes. By implementing capability-based security, Genode enforces security policies. Additionally, Genode enables secure Inter Process Communication (IPC) based on capabilities, while interfaces are defined in an RPC-like fashion: one process announces an RPC server providing functions that can be called by other processes.

TrustZone support is implemented by running the normal world as a Virtual Machine (VM) represented as a process within Genode which is managed by its scheduler just like any other process. The TrustZone VM involves a user-level VMM running on top of Genode, which controls and manages world switches, and handles hypercalls from the normal world. The amount of normal world memory can be configured by Genode and defines the size of available memory of the VM representing TrustZone's normal world. The normal world is based on a standard Debian Linux system including a network interface directly used from normal world. Only slight changes to the Linux system in normal world are required because of security-based restrictions in device usage due to running in normal world of the TrustZone-based architecture.

3.3.2 TrApps Design and System Architecture

The system architecture of the TrApps platform is based on the Genode OS framework including its above described TrustZone support. Due to its micro kernel architecture featuring a small TCB, and the strong capability-based security model, Genode constitutes an approach within the meaning of the objectives of this thesis. Therefore, the TrApps platform incorporates a Debian-based Linux system running in normal world and the *Secure World Manager (SWM)* component running as a Genode process in secure world. The SWM enables a NApp to interact with her associated Trustlet across the TrustZone's world boundary via message queues. In order for normal world applications to make use of TrApps' functionality and communicate with Trustlets, the platform also comprises a Linux kernel module called *TrApps driver* in the following. This system architecture is illustrated in Figure 3.1.

The SWM of TrApps implements all secure world functionality of the TrApps platform, hence, the SWM is responsible for managing the life cycle of all Trustlets. Additionally it handles the interaction of Trustlets with normal world via a message queue abstraction. After uploading a Trustlet to secure world, the SWM instantiates an iso-

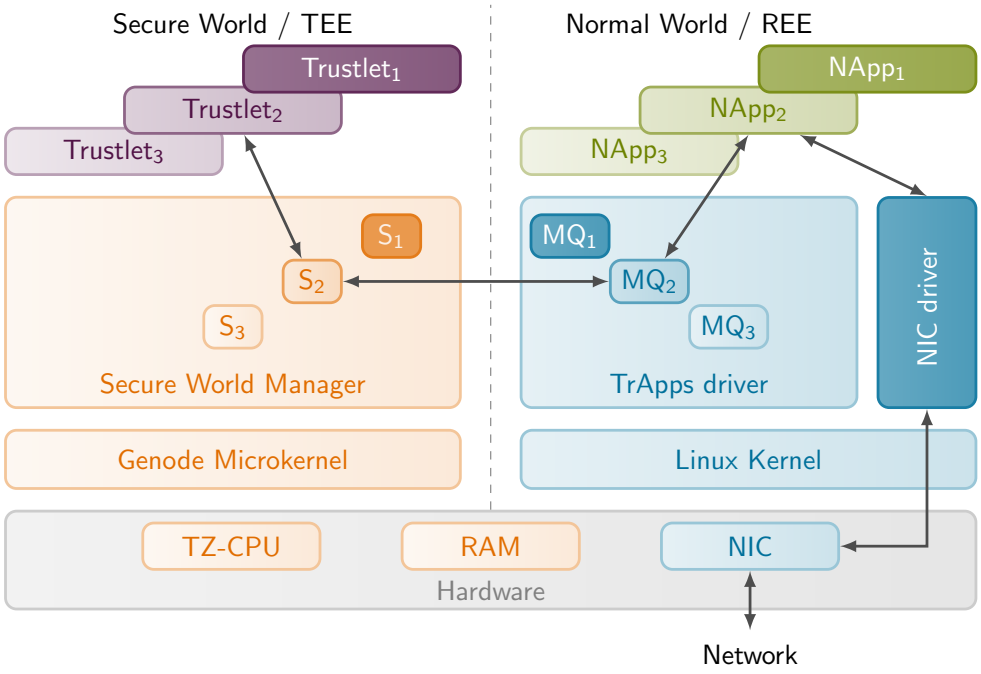


Figure 3.1: TrApps architecture.

lated Genode process and a communication endpoint for that Trustlet to allow it to exchange messages with normal world.

The counterpart of the SWM in normal world is the TrApps driver, which is represented by a virtual device file. The TrApps driver features an interface to NApps that consists of a set of `ioctl` calls that allow the NApp to create a Trustlet in secure world and exchange messages with it. For this purpose, the TrApps driver creates and maintains the message queue abstraction using memory from the kernel space. Interaction of a NApp with the TrApps driver is connection-based, whereas each open connection to the TrApps driver represents one Trustlet in secure world. Hence, Trustlet addressing is done implicitly by the NApp via connection handles to the TrApps driver.

3.3.3 TrApps Cross-world Communication

In order to allow communication between a NApp and its associated Trustlet, TrApps features a message queue abstraction backed by normal world memory as the memory range must be accessible by both worlds. Bidirectional communication is achieved by using a pair of message queues—one for each direction.

The message queue implementation uses cache-agnostic memory (c.f. Direct Memory Access (DMA)), in order to prevent cache conflicts between both worlds since the

secure world and normal world CPU contexts maintain distinct caches. Besides the payload data of the exchanged messages, this allocated memory range holds additional meta data for queue management as well as synchronisation primitives to preserve consistency. For example the meta data assists address translation, since the memory range of the message queue is addressed differently from secure world, kernel mode of normal mode and user mode of normal world.

In order to achieve good performance, the message queue uses explicit world switches in order to notify the other world in a timely fashion. As the usage of the `smc` instruction for explicit world switches is only available in kernel mode, the message queue implementation is part of the TrApps driver. For messages from normal world to secure world, notifications are implemented as hypercalls to the VMM representing the normal world VM in secure world. Notifications in the other direction make use of software interrupts injected into the normal world VM and handled by the TrApps driver.

The message queue features a generic message abstraction, that allows arbitrary memory buffers as messages. Towards normal world, the TrApps driver features an API that allows the creation of a Trustlet, and reading and writing messages to and from the message queue. In secure world, the SWM manages the life cycle of the Trustlet, therefore, during its whole life time a Trustlet can interact with normal world via the SWM which offers read and write methods for messages on the message queues.

3.3.4 TrApps Application Life Cycle and Programming Model

The rationale of applications running on TrApps is that a NApp and Trustlet are developed and deployed together and the NApp manages the Trustlet's life cycle on the TrApps platform. Thus, the life cycle of a Trustlet on TrApps starts with the upload of the Trustlet to secure world. After being uploaded to secure world, the Trustlet can initialise itself before it accepts incoming requests from the NApp. The Trustlet can run for a indefinite amount of time in secure world and process any amount of requests during its life. Eventually the Trustlet gets terminated and removed from secure world by the NApp by closing the connection to the TrApps driver.

The programming model allows a TrApps Application to implement any protocol on the normal world to secure world border. A very simple Trustlet could process every message equally, while a more complex TrApps Application could implement an application-specific protocol on the world boundary of TrustZone, supporting several different message types that are handled differently.

3.3.5 Security and Trust Management

In TrApps, trust is based on the verification of the secure world software stack. This can be ensured by trusted boot of the secure world which allows to control the integrity of the secure world software stack (e.g. Freescale High-Assurance Boot [103]). Thereby, a hash of a public key is burned into the board's fuses which controls that only images signed by the corresponding private key can be booted to secure world. This public key does not belong to the cloud provider, instead it is owned by a trusted third party, which allows splitting the responsibilities and an independent party to control and provide trust into the cloud platform. For example, the cloud provider and the trusted third party could originate from different countries; from a security point of view, the trusted third party is similar to a Certificate Authority (CA). By this, the trusted third party has control over what is booted in the cloud and in secure world in particular. In addition to that, the cloud provider is able to deploy her original complex software stack in normal world. The normal world system is not trusted, and thus, neither part of the above trusted boot nor the TCB of the cloud platform. Especially, the cloud provider does not have to publish the source code of those software components as they are not going to be verified. By inspection of the secure world software before launching the system, the cloud provider can still control that the secure world software has no implemented means to spy on the normal world's contents even though it technically had the ability to do that.

The above described trusted boot, allows the cloud provider, the cloud customers and the cloud users, the verification of the secure world's components. Building on top of that, the Trustlets can simply be verified by a software-based component in secure world that calculates a Keyed-Hash Message Authentication Code (HMAC) of the Trustlet prior to its launch and compares it against the expected one.

Eventually, cloud users must also be able to verify that the cloud platform and the actual application are trustworthy. For example, in case of a web service in the cloud, this can be ensured via Transport Layer Security (TLS): If a TLS connection with a valid certificate issued by the trusted third party can be established, and if the private key for that TLS instance has been uploaded to the trusted cloud application after successful remote attestation, this proves that the cloud platform and the application have passed through all verification steps. Thereby, the cloud users implicitly trust the trusted third party in a similar way as a CA.

As described above, based on secure boot of the secure world software stack, the integrity of the TrApps platform can be protected and ensured. However, in order to achieve confidentiality during sensitive data processing, *secure key storage* is required to

allow the TrApps platform to keep secrets. Therefore, it is required that the trusted platform is able to retrieve a secret value as the basis for key derivation. It is essential that only the trusted platform is able to retrieve that secret value. This can be ensured, for example, by usage of a specially protected persistent memory chip that is only accessible from secure world. Hence, secure storage is dependent on the used hardware platform and could be implemented in various different ways. For a successful deployment of TrApps it is only required that a root key is stored securely by means of *secure key storage*, as this key could be used to derive an arbitrary amount of other keys, for example, one individual key for each Trustlet.

Finally, it is crucial for security that a Trustlet developer understands the trust assumptions in the TrApps platform. A Trustlet must never trust its associated NApp, instead all cryptography must happen inside the Trustlet, while the NApp is responsible for interaction with the network or persisting data locally but only works with data already encrypted by the Trustlet. For example, a full network stack would only be required in the NApp, while the Trustlet does not necessarily need to "understand" the network layer. Instead, the Trustlet is the only entity with the ability to access the plain text of the data, and execute the actual sensitive data processing. A specific example of developing a TrApps Application that runs on TrApps is described in Chapter 4.

3.3.6 Related Work

This section presents existing research works related to the ARM TrustZone technology and the goal of achieving a trusted cloud platform.

Several research efforts use virtualisation technology in order to isolate trusted components that could be as large as complete virtual machines or much smaller fragments of the overall application's logic. Terra [36] protects privacy and integrity of special VMs by means of a trusted hypervisor. Similarly, Proxos [81], Overshadow [26], TrustVisor [77], CloudVisor [124] and Fides [109] follow similar goals and protect privacy and integrity of a VM or secure component from privileged software by isolation. Not all of these approaches protect a complete VM, instead Flicker [75], TrustVisor [77], InkTag [48], VirtualGhost [29] and Fides [109] feature a notion of small trusted environments similar to SGX enclaves. In addition to that, Minibox [66] builds on top of that and proposes a two-way sandbox by combination of the TrustVisor [77] trusted hypervisor with the Google NaCl [123] sandbox. All these approaches establish trusted environments and protect against larger and complex software components compromise such as the commodity OS. Also they aim at a reduction of the TCB as for example the Terra [36] trusted hypervisor with 13,000 Source Line of Code (SLOC) is much smaller

than a full-fledged OS like Linux with several million SLOC.

Virtualisation-based approaches like the ones listed above imply several disadvantages: A trusted hypervisor adds a large privileged software component under the control of the cloud provider to the TCB [102]. Also, management services of the cloud platform comprising maintenance, accounting and monitoring tasks would need to be integrated to coexist with such a trusted hypervisor for usage in a commercial public cloud environment. Furthermore, regular software updates to the hypervisor and other parts of the TCB further complicate operation of a trusted hypervisor in such an environment, as bootstrapping trust into the cloud platform by cloud customers requires them to verify the hardware and software components in the cloud. Finally, providers of commercial public clouds are not interested to use a trusted hypervisor instead of their own ones for several reasons: Certain cloud providers customise existing hypervisors like Xen [14] for their own purposes, and would neither be willing to abandon them nor publish them as it would be required to allow remote attestation. In case of small trusted application components instead of complete VMs [77, 77, 29, 66], the TCB could be notably smaller as those approaches allow removal of a trusted OS from the TCB, even though basic system support is always required in the trusted environment.

Various existing works propose generic platforms for secure component execution. Firstly, Winter et al. [122] propose a trusted computing approach on TrustZone, but opposing our principle of minimising the TCB they use a full-blown Linux kernel in secure world. Nokia proposed On-board credentials [61] which is implemented on M-Shield technology instead of TrustZone and targets mobile and embedded scenarios. In contrast to Winter et al. [122], Luo et al. [70] in fact use a very small secure OS, but their cross-world communication scheme requires the developer to take care of synchronisation of cross-world message exchange. In 2013, Samsung presented the Knox [69] system which uses TrustZone, but it isolates two environments based on Android from each other and targets mobile usage, for example isolating business and personal use of the same smartphone. The Trusted Language Runtime (TLR) by Santos et al. [98, 99] goes beyond the system by Luo et al. [70], offering a more controlled cross-world communication mechanism, however, while their TCB is much larger than ours, they do not support multi-tenant isolated cross-world communication. Jang et al. [54] proposed *PrivateZone*, a system very similar to our TrApps platform. While their system also supports small trusted components it comprises a larger TCB than our TrApps and requires hardware supporting the ARM virtualisation technology. In contrast to the above mentioned trusted hypervisors, the goal of TrApps is the transparent integration into the existing cloud software stack, and support for small manageable trusted components. As described earlier in this chapter, our goal is to enhance the existing software

stack of the cloud provider, even allowing updates to the untrusted software stack that do not affect the secure world, instead of replacing the cloud provider's software stack. Rubinov et al. [96] proposed an approach similar to TrApps, but their system targets an embedded environment with Android running in normal world. This in turn requires all normal world applications to be written in Java and interact with the secure component using Java Native Interface (JNI). Even though TrApps was published earlier, it is more generic in this regard and does not limit the implementation of normal world applications to Java. Also the TCB of their secure OS comprising SierraTEE[2] is significantly larger than the Genode OS used in TrApps. Brito et al. [23] proposed a secure image processing system in a cloud scenario based on the TrustZone technology. In contrast to our TrApps, their approach does not allow general purpose execution and independent and multi-tenant cross-world communication. Finally, Lee et al. proposed a TrustZone-based platform for securing small applets in personal home routers [65]. Apart from targeting cloud scenarios instead of home routers, TrApps features high resource efficiency and flexibility by dynamically scheduling secure and normal world instead of pinning CPU cores statically to one of the two worlds.

In summary, many of the TrustZone-based approaches described above are tailored towards usage on mobile devices and not multi-tenant cloud setups. To the best of our knowledge TrApps [20], at the time of its publication, was the only TrustZone-based secure cloud platform for general purpose Trustlets. Additionally, TrApps features a lean TCB, supports efficient general-purpose execution of secure components with strong isolation in a multi-tenant cloud environment and offers support for flexible and easy to use cross-world communication to application developers. Only after the publication of TrApps, a few other research efforts aimed at achieving SGX-like features in TrustZone [34, 18], establish a generic environment in secure world for shielding unmodified applications from an untrusted OS [44], or virtualising the secure world [50].

3.3.7 Discussion & Conclusion

In this chapter, a trusted cloud platform based on the ARM TrustZone technology has been presented—the TrApps platform. The feasibility of achieving a trusted ARM TrustZone-based cloud platform has been shown and the practical usability will be further detailed in the following Chapter 4, which focuses on the development of partitioned applications that could run on such a trusted platform.

The TrApps platform successfully achieved the goal of providing a trusted cloud platform, as it enables controlling the integrity of the secure world software stack and allows

[2] `https://www.sierraware.com/open-source-ARM-TrustZone.html`

running small Trustlets on top of it that are controlled from normal world. Thereby, with 20,000 SLOC from Genode, and an additional 1000 SLOC from the secure components of the TrApps platform, the overall TCB is reduced by several orders of magnitude when comparing it against a standard Linux-based system comprising more than 20,000,000 SLOC. In addition to reducing the TCB which constitutes a major security benefit, TrApps ensures integrity of the secure world and confidentiality of user data processed by Trustlets in secure world. In the following Chapter 4, with the SGX technology, a transparent memory encryption layer features even stronger protection of sensitive data compared to the TrustZone technology by protection of memory against memory or bus probing.

4 Protecting Applications in the Cloud with Trusted Execution

The previous chapter has introduced the TrustZone-based platform, but has not covered porting applications to run on top of such a platform. Hence, this chapter investigates various forms of equipping applications with a trusted component in order to increase the overall application's security and protect sensitive data being processed by that application. Thereby, this chapter focusses on the usage of trusted execution technologies to offload sensitive business logic to a TEE. This is motivated by the question how real world applications can be deployed on a platform like TrApps as described in Chapter 3, with the specific properties and rationale of a platform like TrApps in mind. The primary goal thereby is the protection of the confidentiality of the sensitive data processed by the application in question. However, the fundamentals are discussed in this chapter in a generic way and are independently applicable to new applications developed from scratch as well as existing applications that are retrofitted to be deployable in such an environment.

In general, applications are split into a trusted and an untrusted component, with the trusted component being deployed in a TEE. Throughout this thesis, this process is called *application partitioning*. The rationale of application partitioning is the identification and extraction of parts of an application's business logic that process sensitive data from that application, and their deployment inside a TEE. This approach aims at achieving higher security by running sensitive components in a TEE. In this thesis, a special focus is set on reducing the TCB in order to increase the overall security, as the security of an application and the successful protection of confidentiality of sensitive data is highly dependent on the amount of trusted code (c.f. Section 3.1.1).

It is assumed that not all of the business logic of an application necessarily requires access to the plain text of the sensitive data but can also work on encrypted data without knowing the plain text. Under this central premise, the untrusted component of a partitioned application can reside outside the TEE and has no access to the plain text. This approach will be called *Trusted Black Box (TBB)* for the remainder of this thesis.

In the following, the goals of application partitioning are described and defined for

the rest of this chapter. Next, an overview over existing approaches of application partitioning and usage of trusted execution technology to secure applications is provided. Following is a section which describes the TBB approach and shows Secure Memcached, an example application built applying that approach on top of the previously described TrApps platform. Afterwards, the TBB approach is applied similarly to two additional distinct applications but using the Intel SGX technology. The two services secured by the TBB approach using SGX are *SecureKeeper*, which is the secured version of the Apache ZooKeeper coordination service, and *Dumbledore*, which is the secured Voldemort key-value store. Thereby, the important metrics that need to be considered when partitioning an application with the TBB approach are discussed.

4.1 Objectives of Application Partitioning

As mentioned above, the goal of partitioning an application is the protection of sensitive data processed by that application. This protection primarily targets the *confidentiality of the data*, and the protection is achieved by encrypting it and processing the plain text of the data only inside a TEE. For this purpose the encryption keys are supposedly only available inside the TEE, hence it is required to establish trust into a TEE before deploying sensitive data or encryption keys to that TEE. However, trust can only be established once the *integrity of that TEE* and the code executed therein is successfully verified. Thus, it is required to protect and verify the software platform's integrity and also to verify the integrity of the underlying hardware platform in order to protect the data confidentiality.

In addition to that, it is a security goal to *minimise the TCB* of the application, as the minimisation of the amount of code that needs to be trusted reduces the probability of exploitable security vulnerabilities of that code (c.f. Section 3.1.1). This motivates the relatively high effort of partitioning an application for usage with trusted execution. Especially this is true if large portions of the code base are not necessarily processing plain text of the data and could remain untrusted. In general, this simplifies verification as all trusted code must be taken into account during verification—even formal verification could be possible for a small enough code base, however, this topic is considered out of the scope of this thesis.

In addition to the above security goals, it is also crucial to partition applications in a way that enables reasonable performance of the resulting trusted application. At a certain point the cost of added security would be just too high. An important aspect to be considered in this regard, is the border between the trusted and the untrusted part of the partitioned application. Switching the execution mode from trusted to un-

trusted (and back) induces a notable performance hit [121, 70], and thus, switching too frequently should be avoided.

Furthermore, all data moved across the border of a TEE and the outside untrusted environment must be checked rigorously to prevent exploitation of vulnerabilities by crafted inputs from the outside [24]. This also applies to the number and signatures of functions calling into (and out of) the TEE, as each additional function increases the attack surface of the trusted application.

Especially in a cloud setting it is an essential requirement for applications to support multi-tenancy and to be able to isolate tenants securely from each other. Individual tenants should be able to simultaneously use a (cloud) service or application while getting the impression they are the only user by preserving *performance isolation*. In addition to that, tenants should not be able to access other tenant's data but only their own, which forms a security-related aspect of isolation.

4.2 Trust Model and Assumptions

This chapter covers the partitioning of applications for usage in an untrusted cloud setting. This means, that the Cloud Provider who owns the hardware where the application is deployed, is neither trusted by the Cloud Service Provider nor the Cloud User of the cloud application. Hence, the Cloud Provider must be detained from accessing or modifying the data processed by the service without being noticed. Therefore, the data must be encrypted and basic integrity-protection (integrity protection is covered in more detail later) should be applied whenever the Cloud Provider could access it. Additionally all sensitive software components deployed in the cloud must also be integrity-protected at least in a basic form. Otherwise it could not be guaranteed that the encryption is actually executed and trust into the software platform could not be established. In this context, sensitive software components are components that may have access to the sensitive data and comprise not only the application deployed by the Cloud Service Provider itself, but also underlying infrastructure components such as the hypervisor and the OS.

In this thesis, it is assumed that the *Cloud User* trusts the *Cloud Service Provider* that developed and deployed that service in the cloud, as opposed to the *Cloud Provider* which is not trusted. However, the *Cloud Service Provider* and the *Cloud User* do not necessarily need to be two distinct entities, instead the *Cloud Service Provider* may use that software herself. Finally, in this context we require the basic properties of secure cloud platforms as defined in Section 3.2 to apply here as well.

Partitioning in this chapter focuses on *data-handling services*, meaning that the ser-

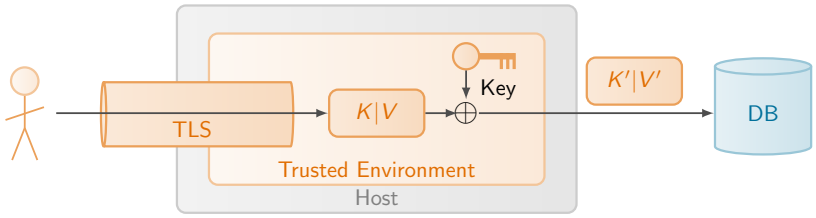


Figure 4.1: TBB approach

vices or applications that are partitioned mostly store, receive and return data, and only perform limited server-side processing on that data. Exemplary for data-handling services, in this thesis the key-value stores *memcached*[1] and *Voldemort*[2], as well as the ZooKeeper coordination service [51] as a more complex cloud service have been chosen for application partitioning and are introduced in the following sections of this chapter.

In order to protect the confidentiality of the sensitive data processed by the trusted component, the integrity of the trusted component and the underlying hardware platform must be guaranteed. This requires at least basic integrity protection of the trusted component, defending against malicious code modifications or emulation of the trusted execution technology. Protection of TEEs against replay and rollback attacks is not covered in this thesis but has been investigated for example by Matetic et al. [74].

4.3 The Trusted Black Box Approach

The goal of the Trusted Black Box (TBB) approach is to split the code base of an application into an untrusted and a trusted part. The untrusted part has no access to the plain text of the sensitive data at any time and processes the data only in encrypted form—thus the name Trusted Black Box (TBB) approach. The basic principle of the TBB approach is exemplary illustrated for a key-value store application in Figure 4.1.

With the TBB approach, for example, a network stack implementation could remain untrusted, as long as a TLS connection on top of it terminates inside the trusted environment. In contrast, the trusted part is the only party that has access to the encryption keys, and hence is able to decrypt and process the sensitive data. Thereby, the trusted part is supposed to be as small as possible in order to decrease the TCB and the attack surface of the resulting partitioned application. However, the TCB is not the only facet to be optimised, also the performance of the application is crucial. Therefore, due to the significant cost of changes of the execution mode between trusted and un-

[1] Memcached `http://memcached.org/`
[2] Project Voldemort `https://www.project-voldemort.com/voldemort/`

trusted [11, 121], the border between trusted and untrusted code components has to be smartly chosen. For example, it might be useful to slightly increase the TCB in favour of much lesser execution mode changes.

Confidentiality of the sensitive data processed by the partitioned service is protected by encryption with an encryption key only known to the trusted component. For this purpose *data sealing* (c.f. Section 2.2) mechanisms can be used, as they allow the deterministic retrieval of an encryption key derived from the TEE's identity. This guarantees that only a trustworthy TEE instance can access that key and decrypt the sensitive data, and allows the secure storage of sensitive data outside the TEE to survive TEE restarts.

A TEE developer must be aware that copying data across the TEE's boundary is potentially risky. Data copied from the outside into the TEE could be arbitrarily modified by the untrusted code outside and the TEE must implement appropriate measures to check all data from the outside. Thereby it should be ensured that the TEE does not crash due to unexpected inputs, but more importantly that it does not leak sensitive data. Furthermore, the TEE must execute credibility checks of all incoming data, for example even return values from system calls can not be trusted by an SGX enclave as the whole OS is untrusted [15]. On the opposite side, the TEE must carefully ponder what data is externalised and in what way. Therefore, the TEE developer has to take measures to prevent unintended data leakage of sensitive data from the TEE to the untrusted environment. Furthermore, it must be ensured that data that is externalised intentionally is protected adequately in order to guarantee the required confidentiality and integrity properties of the sensitive data.

Protecting the integrity of sensitive data in the TBB approach has several distinct facets. The integrity of the data in itself can be protected by an HMAC over the data. The crucial properties of an HMAC in this case are the guarantee that it can not be forged by malicious entities as it requires a secret key, and it allows detection of any changes to the protected data.

However, higher level integrity is not implicitly protected by this, for example swapping the (encrypted) value of a key with another (valid) value is not directly detectable, if not the connection between key and value of a key-value pair is protected as well. Also, freshness of the data can not be guaranteed and rollback or replay attacks can not be detected or prevented by this, for example, swapping a value with an older correct value of that same key is not detected by this form of integrity protection. Problems like those have been investigated for example by Brandenburger et al. [17] and are considered out of scope of this thesis.

As mentioned above, it is also essential for the owner of the sensitive data to be able to verify the integrity of the platform and the trusted component. Otherwise trust can not

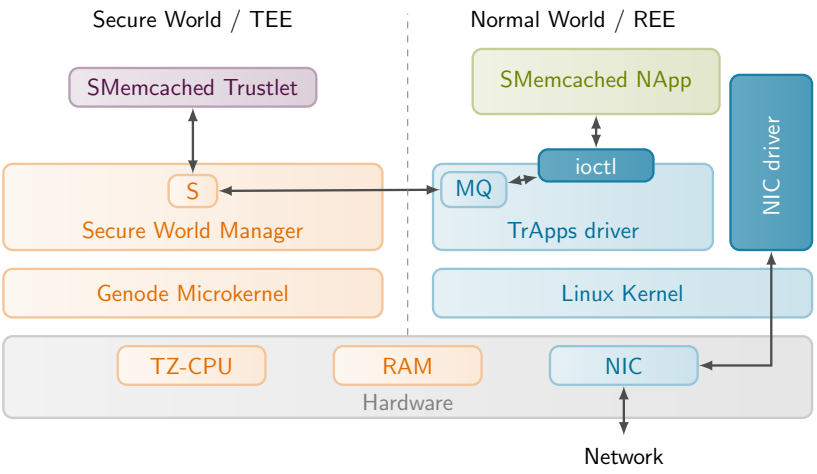


Figure 4.2: Secure Memcached on TrApps system architecture.

be established into that trusted component, and sensitive data should not be confided to it. Therefore, a verification including a proof of the integrity of the trusted component and the underlying hardware platform must be provided to the data owner before she can transfer the sensitive data to the trusted component. This proof is accomplished by means of a *remote attestation* procedure which has to be supported by the trusted execution technology that is used. Given the integrity of the platform could be verified, the TBB approach allows the protection of the confidentiality and basic integrity of the sensitive data, while allwing to execute general-purpose processing on that data.

4.4 Secure Memcached on TrApps

This section describes the partitioning of the *Memcached* key-value store[3] to run on top of the TrApps platform. Memcached is an in-memory key-value store written in the C programming language. Partitioning of this application follows the principles defined in the previous sections with the goal of protecting the confidentiality of the data stored in Memcached. Furthermore, this use case application showcases the functionality of the TrApps platform and delivers a proof of concept, also it provides an estimate of the possible performance of the TrApps platform.

4.4.1 Design and Implementation of Secure Memcached on TrApps

The system architecture of Secure Memcached is illustrated in Figure 4.2, which shows that the partitioning of Memcached to run on TrApps leads to a trusted and untrusted component supposed to be deployed in secure and normal world of TrustZone respectively. The communication between those two components across the TrustZone's world boundary is processed via the TrApps driver (c.f. Chapter 3).

According to the principle of reducing the TCB, as a starting point all business logic of Memcached is considered untrusted. Then, code that can only work with plain text is identified and offloaded to a TEE. The general idea now is to protect the client-server communication—for example by using TLS, the state of the art technology for this purpose—while terminating that encryption only inside the TEE on the server side. This ensures that plain text is never available outside of the TEE (unless the TEE decides to hand out the data). Then, the trusted component analyses the incoming data and, in case of the Memcached example, identifies keys and values of key-value pairs as the sensitive data that is to be protected.

The most important and most common requests to a key-value store are `get()` and `set()`, whereas a `get()` request will only contain the key whose value is being requested, and `set()` contains key and value which is to be stored by the server. In order to protect the data, the TEE encrypts the plain text key and value of the key-value pair with a symmetric encryption mechanism using a secret encryption key only known to the TEE and forwards the encrypted data to the actual (untrusted) database.

Without server-side operations that need to be executed on the plain text data, the whole service could reside in the untrusted environment and clients could be adjusted to encrypt the data before interacting with the server. However, this would increase the complexity of key distribution and management, because in that case all clients would need to agree on a shared encryption key and future exclusion of malicious clients is much harder. However, apart from the `get()` and `set()` operations, Memcached also supports `increment()` and `decrement()` operations. In order to execute an `increment()` (or `decrement()`) operation, the server needs to know the plain text of the data. Thus, Secure Memcached encrypts key and value data from `get()` and `set()` requests. This allows to call into the trusted component during the processing of an `increment()` or `decrement()` operation, decrypt the data, execute the operation on the plain text and encrypt the altered data before leaving the TEE. Hence, during the execution of the `get()` and `set()` operations the trusted component has to encrypt key and value of all incoming requests, respectively.

[3] Memcached `http://memcached.org/`

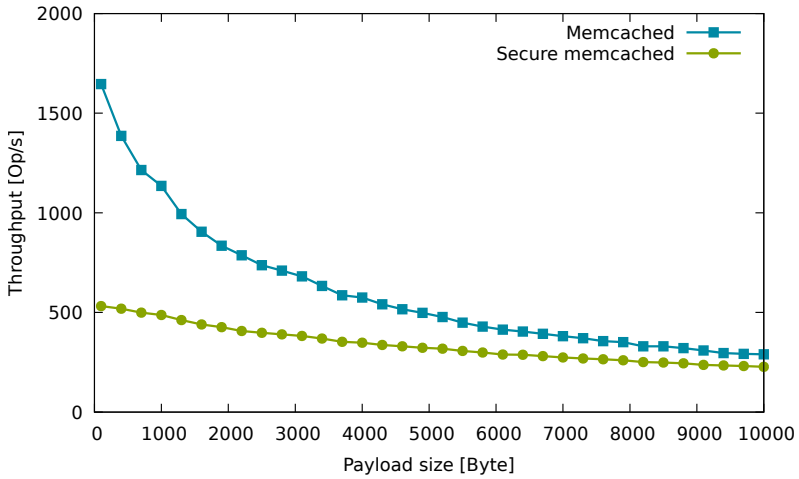


Figure 4.3: Throughput of Secure Memcached on TrApps.

As mentioned above, usage of TLS as opposed to a shared secret that all clients need to know also poses the advantage of simpler and more flexible key management. Clients could connect to the server and a secret encryption key is negotiated for each connection individually. The TEE is the only component that has access to the symmetric encryption key of all encrypted data stored on the server. This makes it easy to integrate access control mechanisms into the TEE and to exclude rogue clients that have been benign and turned malicious later on. Because the clients never knew the encryption key of the data, they can not extract any information from the encrypted data on the server if they got a chance to access it. Only data a client has actually accessed during the time she was permitted to access the server is known to that client. In addition, this approach removes the necessity to have a key distribution mechanism for the encryption key between all benign clients. Clients only need to be able to establish an authentic TLS connection with the server, therefore it is crucial to ensure authenticity of the server's TLS certificate.

4.4.2 Evaluation of Secure Memcached on TrApps

We have measured the impact of using TrustZone-enhanced security on the basis of TrApps in the form of the TrApps-secured Memcached in-memory key-value store application. For the evaluation we started the official Memcached benchmarking application *memslap* on a remote host and issued requests to the service. The TrApps-secured memcached service has been executed on the NXP i.MX53 development board[4] running

[4] c.f. `https://www.nxp.com/design/development-boards/i.mx-evaluation-and-development-boards/i.mx53-quick-start-board:IMX53QSB`, last accessed 09/2020.

Genode in secure world and Debian in normal world (c.f. Section 3.3.1).

Figure 4.3 shows the throughput of Secure Memcached running on TrApps compared to a regular Memcached instance without security running in normal world on the same hardware in order to allow fair comparison. As can be seen, with increasing payload sizes the relative overhead of the increased security with TrApps decreases. With very small payload sizes up to 1000 Bytes, the overhead is quite huge with about 62% due to the constant world switch overhead between secure and normal world. With larger payload sizes (payloads $>$ 1000 Bytes are considered *large*) the relative overhead of the world switches becomes more and more insignificant and the dynamic overhead of the cryptography required to decrypt and encrypt the messages is acceptable with 34%.

4.5 Adding Transparent Encryption of Memory by using Intel SGX

The previous sections have shown the feasibility of partitioning an application to run on top of TrApps using the ARM TrustZone technology and protect sensitive application data successfully. However, this solves the overall problem only partly according to the defined goals in Chapter 3.

While ARM TrustZone can protect against bugs in the normal world software stack and prevents the cloud provider from booting arbitrary system images unnoticed, the memory of the machines in the cloud is still vulnerable to physical attacks such as the cold-boot attack. For this purpose, in this thesis the Intel SGX technology is investigated as well, featuring a trusted execution technology with additional transparent memory encryption and integrity protection.

Memory encryption of Intel SGX is transparent to the user in the sense that no encryption keys need to be managed directly and the encryption and decryption operations are done in the background implicitly by the MEE. From the user's perspective, the encrypted enclave memory appears unencrypted when being accessed from a valid enclave context and can be used just like regular memory. A valid enclave context denotes that the memory accesses originate from a valid enclave and are targeted to memory ranges that are part of that same enclave's ELRANGE. The only perceptible difference is the memory access latency, which is higher compared to regular insecure memory due to the required cryptographic operations done transparently by the MEE [58]. Privileged accesses directly to the protected memory ranges of the enclave, for example by the OS which is untrusted in the SGX trust model, are prevented by the SGX-capable CPU. In such cases, instructions accessing the memory just return `0xFF` (c.f. abort page semantics in Section 2.4.2) regardless of the actual contents of that memory range. However, hardware attacks such as the cold-boot attack reading directly from

the memory cells return the encrypted memory contents. By this the memory contents of an enclave, comprising the code, data, stack and heap memory ranges of the enclave, receive strong cryptographic protection. However, the risks of interaction with the outside world must still be respected by the enclave developer, and it must be ensured that the enclave does not externalise sensitive data accidentally.

In addition, sensitive data that is intentionally stored outside must also be protected against rollback attacks [17]. As the untrusted OS has the power to terminate and re-instantiate an enclave at any given time, an enclave has to make sure to detect stale data input when accessing previously externalised data to untrusted storage. In order to solve this, trusted monotonic counters could be used for example that allow the detection of stale data by the enclave. However, this requires to trade off the frequency of externalising data using trusted monotonic counters versus the cost of using the trusted monotonic counter and the granularity of the externalised data checkpoints. As this raises many questions orthogonal to the research questions of this thesis, this kind of attack is considered out of scope of this thesis.

Currently Intel SGX only allows the reservation of a relatively small amount of memory for usage as transparently protected EPC (c.f. Section 2.4.2). Also, this memory range is shared between all SGX enclaves on a hardware platform. Once the available EPC memory is exhausted, the costly *SGX paging* has to be executed, which induces a significant performance impact as it requires a re-encryption of all pages.

We have evaluated the significance of the performance penalty of EPC paging using a custom key-value store application inside an enclave. The measurement shows the performance of a simple key-value store versus the same key-value store running inside an enclave for increasing sizes of the memory working set. It can be seen in Figure 4.4, that as long as the working set fits into the available EPC the performance of the enclaved key-value store is relatively close to native performance. In that case, the performance difference stems from enclave entries and exits and the MEE encryption. As soon as the working set exceeds the available EPC, however, the performance of the enclaved key-value store immediately drops quite significantly due to the EPC paging that is necessary to fulfil the increased memory demand of the application in the enclave. In an evaluation of this kind it is important to consider the actual working set memory footprint of the enclaved application as opposed to the total accumulate of memory allocations. It is not crucial how much memory an enclave allocates in total, as unused memory portions can be evicted once and do not occupy valuable EPC memory unless the memory region is touched thereafter. Important is the total accumulate of memory regions—strictly speaking, the number of memory pages as EPC paging works at page granularity—that are actually used by the application in a given time span. In order to

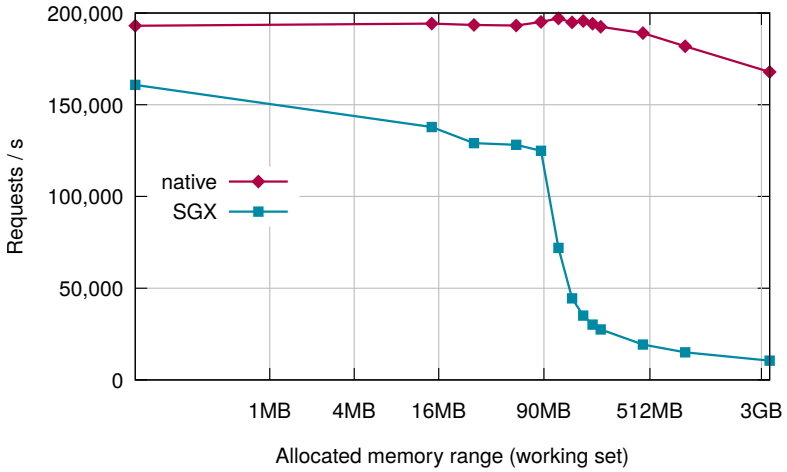


Figure 4.4: SGX paging overhead.

respect this in the benchmark, the number of key-value pairs of known size is increased and each of them accessed randomly, and by this at approximately the same rate. This controls the size of the working set memory footprint of the key-value store application, and thus, the amount of memory that is actually in use by the enclave to pointedly trigger EPC paging. The results of this benchmark in Figure 4.4 clearly show that EPC paging should be avoided when possible to increase the performance of the enclaved application and that the effect of EPC paging on the performance is quite significant.

4.6 SecureKeeper Coordination Service

This section describes the application of the application partitioning principle according to the above described TBB approach to the Apache ZooKeeper coordination service in order to secure the application and protect the user data with an integrated trusted component. Hereby, ZooKeeper is a service that implements coordination primitives for distributed applications and intended to be used by distributed cloud applications as a backbone, representing an essential building block for coordination of distributed application components.

4.6.1 Apache ZooKeeper

ZooKeeper [51] is a coordination service that is supposed to be used by distributed applications to offload their coordination tasks to a central service deployed in the cloud. This prevents repeated re-implementation of coordination primitives for several appli-

Operation	Description
`create(path, [payload])`	Create a znode in the data tree
`delete(path)`	Delete a znode from the data tree
`setData(path, payload)`	Set the payload of a znode
`getData(path)`	Get the payload of a znode
`getChildren(path)`	Get child znodes of a znode
`exists(path)`	Check existence of a specific znode

Table 4.1: ZooKeeper simplified operations overview.

cations, and instead makes coordination usable and integrable by distributed applications without them implementing those themselves.

ZooKeeper stores all its data primarily in memory, thus, it is not supposed to be used as a database for large chunks of data. In contrast, it offers high performance for small amounts of data, and features high consistency guarantees to the clients. ZooKeeper maintains a tree of so called *znodes*, that are identified by a *path* like in a file system and can store a small amount of arbitrary payload data.

From a client's perspective, ZooKeeper resembles a key-value store: it basically provides `get()` and `set()` operations for accessing and altering znode's and their payload data. Payload data could for example be the configuration of a distributed system, stored in a set of hierarchical ZooKeeper nodes.

Data is maintained in ZooKeeper internally as a tree of znodes. Hence, znodes can have children and maintain a relation to their parental znode. All znodes are basically equal in the sense that they can have children and store payload data at the same time. However, znodes can be created with the additional attributes `ephemeral` and `sequential`. Ephemeral znodes are linked to the Transmission Control Protocol (TCP) connection of the client that created them, and are automatically removed by ZooKeeper once the client connection terminates—either intentionally or by time out. Sequential znodes are equal to regular znodes, however, upon their creation ZooKeeper appends a monotonically increasing counter to the znode's path name. The two flags can be combined, so znodes can be sequential and ephemeral nodes at the same time. In addition, znodes can be *watched* by clients: a client can register for being notified by ZooKeeper about any changes to that znode. Once a change to a watched znode happens, ZooKeeper will inform the client about that change, and the client can execute further operations based on that knowledge; for example read the new payload of that znode.

In addition to the above mentioned `get()` and `set()` operations, the ZooKeeper API allows to retrieve and alter the payload of a znode (`getData()` and `setData()`). Further-

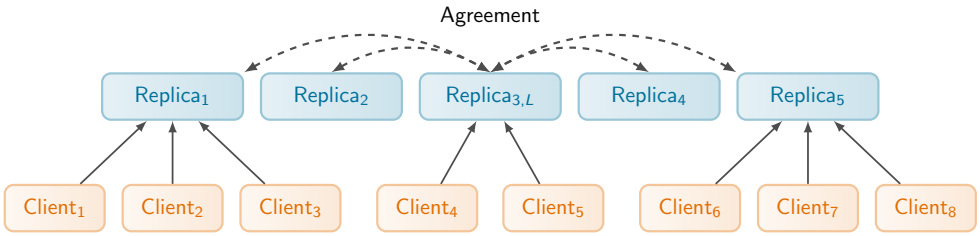


Figure 4.5: Apache ZooKeeper coordination service architecture.

more, a list of the children of a znode can be requested by issuing the `getChildren()` operation, and the existence of a particular znode can be checked with `exists()`. Table 4.1 shows an overview of the most common ZooKeeper operations.

ZooKeeper is a distributed system itself, with its architecture consisting of a number of ZooKeeper replicas that form a ZooKeeper cluster. Its architecture is illustrated in Figure 4.5. One replica is the designated leader of the ZooKeeper cluster and negotiated by all replicas during cluster start up or upon leader failures. The ZooKeeper leader is responsible for maintaining a global order on all write accesses to the ZooKeeper data tree. The other replicas, the ZooKeeper follower replicas, receive state updates from the ZooKeeper leader and maintain a full copy of the data set. Thereby the communication and agreement between ZooKeeper replicas follows the ZAB [56] protocol.

ZooKeeper follower replicas are able and allowed to respond to read-only client requests directly without asking the leader. In contrast, write accesses are always forwarded to—and brought into a consistent global order by—the leader replica. In addition to the global write order, ZooKeeper also guarantees a local first in, first out (FIFO) order of all (read and write) requests of a single client.

Due to the replicated nature of ZooKeeper, the service can tolerate crash faults of a minority of replicas and stays operational with a majority of correct replicas (tolerates `f` failures with `2f+1` replicas). Upon leader failure, a new leader is negotiated by the remaining functional ZooKeeper replicas. ZooKeeper also persists its data on disk using snapshots of the in-memory database and a write ahead log of committed operations.

ZooKeeper Use Cases

ZooKeeper can be used for arbitrary coordination tasks. A very simple one is *configuration management*, where participants of a distributed system store the system's configuration inside the payload of ZooKeeper's znodes. A master node could write configuration values to specific znodes while a set of participants of a distributed system watch those znodes and read the currently active configuration values from it.

Group membership of a distributed system is another example for a coordination primitive that can be implemented with ZooKeeper. The availability of members of a distributed system can be described by the existence of a specific znode in the ZooKeeper's data tree that each member of the group creates for herself. The znode could for example be named after the hostname of the member that it represents. Using the above described ephemeral flag makes sure that the znode can only exist while the client is still alive because otherwise ZooKeeper would remove that znode from its data tree. This allows other participants of the group to retrieve a list of alive replicas by issuing the `getChildren()` operation to the parent znode under that all members are inserted.

Finally, another example for a coordination task executed by ZooKeeper is *leader election*. The goal of this coordination primitive is for a set of replicas to negotiate a common leader replica that all replicas agree upon. This can be implemented as an enhanced variant of the above group membership primitive. For leader election, all replicas create a znode representing themselves under a common parent znode with an additional sequential flag. This way, the group can agree that the currently active (and alive due to the ephemeral flag) leader is the one replica with the lowest sequence number. This guarantees that all group members agree on the same leader and that a leader failure is detected by the other replicas triggering a new leader to become active.

4.6.2 ZooKeeper Privacy Proxy

In order to provide a secure ZooKeeper service that protects the confidentiality and integrity of znode names and payload by usage of trusted execution and following the above principles, the service needs to be partitioned. As a first step towards the full secure coordination service SecureKeeper, we built the ZooKeeper Privacy Proxy (ZPP).

During the design of the ZPP, all sensitive data that is processed in the ZooKeeper coordination service and parts of its business logic that require access to that data have to be identified. In case of ZooKeeper, the sensitive data primarily comprises the znode path names and the arbitrary payload data stored inside a znode's payload field. Thereby, request processing is critical business logic as it requires access to the plain text of the sensitive data. For example, the `getChildren()` operation must know the znode's name in order to identify its child nodes. In contrast, the ZooKeeper component that persists data to disk can work with encrypted data without ever knowing the plain text, and thus, can stay outside the TCB.

The rationale of processing sensitive data of ZooKeeper in a TEE is to insert the ZPP TEE in between the connection between clients and the ZooKeeper replica. Clients are supposed to establish an encrypted connection that terminates inside the TEE to pro-

tect the data during network transmission. Once the data arrives inside the TEE, the transport encryption can safely be removed because the data is now protected by the TEE mechanisms. Next, the ZPP starts parsing the decrypted incoming message and identifies znode names and payload inside it. This data is encrypted with a secret key only known by the TEE and replaces the original values in the source message. Afterwards, the ZPP forwards the altered message to the outside untrusted component that continues message processing without knowing the particular plain text of the znode's path and payload but working with the respective cipher text. This allows to keep large parts of the application untrusted and minimise the TCB of the final application.

The ZPP opens a network socket where it listens to incoming connections from ZooKeeper clients. Upon incoming requests of a client, the ZPP opens a new connection to the ZooKeeper cluster and forwards the message after processing to a regular ZooKeeper replica. With this minimally invasive integration, the ZPP acts as a proxy that mimics ZooKeeper clients for the ZooKeeper replicas and ZooKeeper replicas for ZooKeeper clients. Both ZooKeeper replicas and ZooKeeper clients see the ZPP as their ZooKeeper protocol-compliant communication partner. The socket-based integration of ZPP even enables it to run on a distinct host and renders it completely independent from the used trusted execution technology. In fact, ZPP could even be deployed on a (virtual) host that is only "declared as trusted", and executed without actual trusted execution, hence it demonstrates what parts of ZooKeeper's business logic must be trusted and how partitioning of an existing service works.

ZPP encrypts znode payloads using a secret key K_s that never leaves the ZPP's trusted environment but is shared between all ZPP instances. The latter is required in order to allow users access to znodes created by other users, an essential basic functionality of ZooKeeper in order to implement coordination primitives. Thereby, the encryption of znode payloads—called *payload encryption* in the following—is implemented with a fast symmetric encryption mechanism.

Paths of znodes are also encrypted by the ZPP—called *path encryption* in the following—in order to protect the path names of znodes as they are essential in a ZooKeeper environment. For example, some use cases test for the existence of specific znodes (with empty payload), thereby the only sensitive information is the znode's path itself. Path encryption is performed for each part of a path delimited by a slash character individually. After encryption, the encrypted path components are additionally BASE64URL-encoded in order to prevent usage of unprintable characters and the slash character in particular in an encrypted path name that ZooKeeper could otherwise not handle.

The above described path encryption approach works well in most cases, however, sequential nodes pose a very specific problem here and require special handling. The

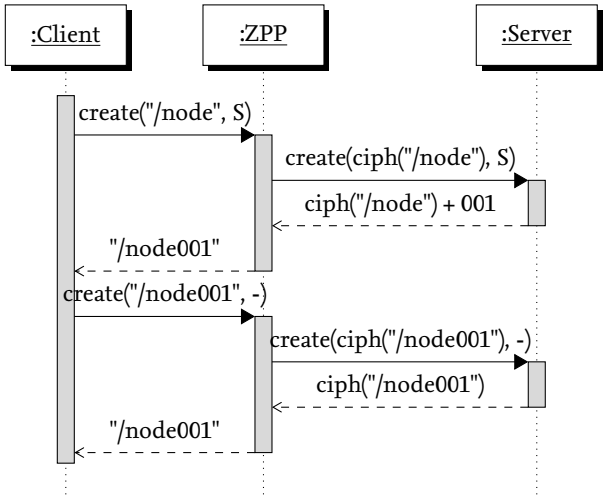


Figure 4.6: Node name collision situation when creating sequential nodes. *Sequential flag is denoted as "S", no flag as "-" and the ciphertext of "x" as "ciph(x)".*

problem is illustrated in Figure 4.6: in case a sequential node is created, ZooKeeper automatically adds a monotonically increasing number to the path name and returns it to the user as a return value of the `create()` operation. However, if a user creates a regular node with a path name that ends with a number that looks like a sequence number a conflict arises as this could not be distinguished from a real sequential number and could even clash in cases where the sequence number given by ZooKeeper is the same number the users chose as a regular znode's path name.

The approach of ZPP is to store additional meta data in so called *dictionary nodes* to cope with this problem. Essentially, the dictionary znodes are regular znodes in a special path name space only visible to ZPP instances but invisible to normal ZooKeeper clients. Those dictionary nodes are used to store the current sequence number's value of all nodes, except for ephemeral znodes as they could not have any children, and leaf nodes as they do not have any children (yet). The dictionary itself is implemented as a hash table stored in the payload of a dictionary node and held in memory by all ZPP instances for performance reasons—essentially storing the data in memory prevents an additional network round trip for lookups. However, all ZPP instances must watch the dictionary nodes in order to keep track of any changes to them. This is implemented using the ZooKeeper watch feature on the respective dictionary nodes.

For each `create()` operation (and `delete()` operation), the respective sequence number of the parent node has to be altered. This is done by issuing a `multi()` operation combining the `create()` or `delete()` of the znode to be created or deleted with a `setData()` operation that updates the payload of the dictionary node with the new

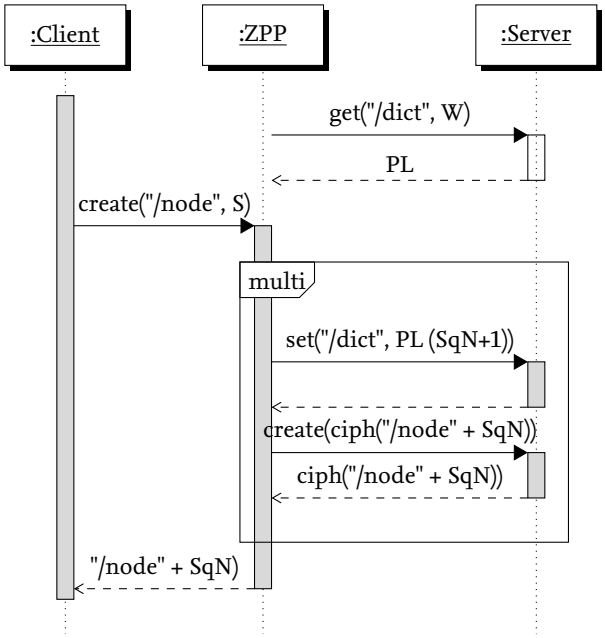
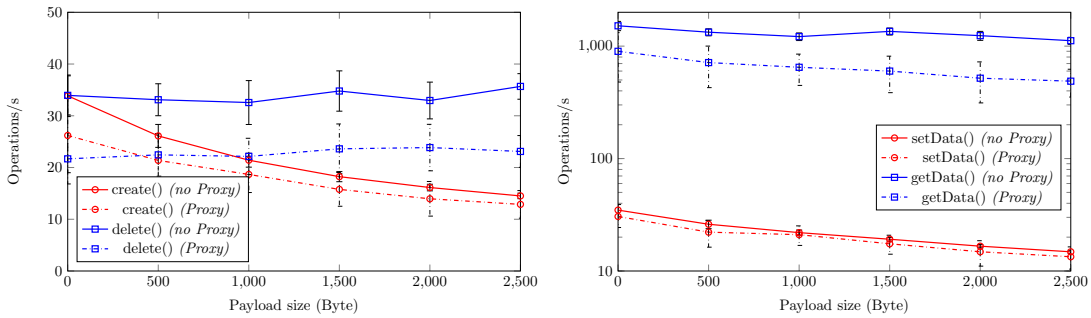


Figure 4.7: Node creation procedure involving sequence numbers from dictionary nodes. *We denote a watch callback registration as "W", the dictionary's payload that will include the sequence numbers as "PL", and the sequence number of the parent node of "/node" as "SqN".*

sequence number. This `setData()` update is also conditional, thus, it fails in case of conflicts with other parallel operations causing the whole `multi()` operation to fail and ensuring consistency of the data in ZooKeeper. By using the dictionary nodes, the above outlined problem of conflicting sequence numbers can not happen and the secured ZooKeeper variant still behaves like original ZooKeeper instances. Figure 4.7 illustrates the process and the interaction of ZPP instances with a dictionary node.

A ZPP instance maintains an individual connection to a ZooKeeper replica for each client connected to it. By this, failures of client connections can be relayed to the ZooKeeper replica in order to support the original behaviour for ephemeral znodes. A ZPP failure is experienced by a ZooKeeper client the same as a regular ZooKeeper replica failure, causing the client to choose a different replica and establish a new connection. However, a client would choose a different ZPP instance instead of a different ZooKeeper replica in this case. Similarly, the ZPP can switch to another ZooKeeper replica in case of replica failures which leads to a deterministic behaviour in all cases of possible failures.



(a) `create()` and `delete()` operations.

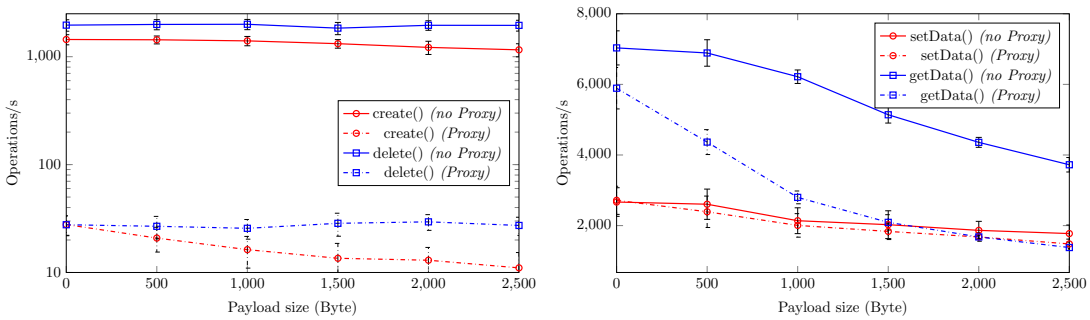
(b) `set()` and `get()` operations.

Figure 4.8: Throughput of ZPP for synchronous operations.

Evaluation of ZooKeeper Privacy Proxy

Since the TCB of a secure application is crucial for its security, we measured the amount of trusted code in a ZooKeeper installation enhanced by a ZPP. For a secure ZooKeeper installation using ZPP instances, the ZPP's source code is the sole TCB of the full application, excluding the ZooKeeper's source code as it is untrusted and by this reducing the amount of overall TCB. Compared to the ZooKeeper server's official code base of around 85,000 SLOC of Java code, using the ZPP with less than 4,000 SLOC reduces the amount of trusted code by more than a factor of 20× and by this increases security significantly. Note that, in addition the ZPP is written in native C Code and does not require the complexity of the Java runtime underneath.

Another crucial property of the secure ZooKeeper with ZPP is the overall performance of the secured service as experienced by its clients. Due to the generic nature of its concepts and the integration via network sockets, ZPP is independent from the actual trusted execution technology and could be deployed on ARM TrustZone using a platform like TrApps, in SGX enclaves or on standalone systems "declared" as trusted (for example because they are located in a physically protected room). This section shows an evaluation of running ZooKeeper replicas, ZPP instances and ZooKeeper clients each in their own VM on an OpenStack [104] IaaS cloud in order to measure the performance implications of using the ZPP. All VMs ran Ubuntu 13.10 and were equipped with 2 virtual CPUs and 2 GB memory for the ZooKeeper replicas, and a single virtual CPU and 512 MB memory for the ZPP instances. In order to minimise the impact of background noise induced by other VMs running on the same cloud platform, requests were issued in batches and the measurement results show the average value of 15 repetitions for each data point. The individual operations were executed synchronously and with different payload sizes ranging from 0 to 2.5 KB.



(a) `create()` and `delete()` operations.

(b) `set()` and `get()` operations.

Figure 4.9: Throughput of ZPP for asynchronous operations.

The benchmark evaluates the individual operations of ZooKeeper and compares the performance of the original ZooKeeper versus the ZooKeeper installation equipped with the ZPP. The results of synchronous operations are illustrated in Figure 4.8a and 4.8b. As can be seen, the performance of a ZPP-secured ZooKeeper is lower but still close to the initial performance of a plain ZooKeeper installation. For all operations except `delete()` the payload of the respective znode is transmitted either in the request or in the response, therefore, the performance is dependent from the payload size. In addition to that, it can be seen, that the performance of `getData()` is much higher than all three other operations, because this operation is read only and does not require ordering on the ZooKeeper leader but can be answered by the ZooKeeper replicas directly. Even in this case, during usage of the ZPP the performance is close to the regular ZooKeeper's performance. In total we measured an average throughput degradation by the enhanced security when using ZPP of approximately 27%.

In addition to synchronous requests as described above, ZooKeeper also supports asynchronous requests. While this is supported by ZPP-equipped ZooKeeper instances as well, usage of the ZPP induces a major performance impact for such requests in some cases. As `create()` and `delete()` operations require access a dictionary node in order to keep the sequence numbers up to date, the request can not finish before the ZooKeeper replica has acknowledged the dictionary update. If the acknowledgement is not waited for, the ZooKeeper database could end up in an inconsistent state. However, this implies that asynchronous requests are processed synchronously solely by forwarding them through the ZPP. While this is transparent to the clients it leads to a significant performance impact for asynchronous `create()` and `delete()` operations compared with the baseline insecure ZooKeeper. This effect can be seen our evaluation graph in Figure 4.9a. The `get()` and `set()` operations are not impacted by this problem because no dictionary updates are required for those operations. As can be seen

in Figure 4.9b the performance of a ZPP-equipped ZooKeeper is close to the insecure ZooKeeper's performance.

This section has shown how the partitioning of an existing application's code base for usage with trusted execution technology works in general and that it has the potential to remove a large amount of code from the TCB of a secured application. The achieved security guarantees of ZPP thereby focus on the confidentiality of the processed data, but also protect their integrity from unintended or malicious altering in a basic form.

4.6.3 SecureKeeper

In the previous section, the ZPP was described as an initial solution to secure the data processed in a ZooKeeper cluster. Based on that, this section discusses *SecureKeeper*, an SGX-based secure ZooKeeper implementation on the basis of the above ZPP but integrated directly into the ZooKeeper replicas as SGX secure enclaves.

Like all Java-based application ZooKeeper can only run with the help of a Java Virtual Machine (JVM) underneath. However, unless the JVM is placed inside the secure environment, the execution of Java code can not be trusted. This increases the complexity of securing any Java-based application by trusted execution, because it is neither trivially possible nor reasonable to run a full JVM inside a TEE due to the following reasons. Firstly, it would require a significant amount of system calls inside the trusted environment to function properly. Providing this level of system support inside a TEE in addition to the JVM itself is a rather complex task and increases the complexity and TCB of the overall application significantly. For example, the commonly used OpenJKD [5] alone comprises more than 6,000,000 SLOC. In addition to that, the deployment of a full-blown JVM especially in an SGX-based TEE, leads to a very large trusted memory footprint of the final application, which leads to substantial performance penalties due to the required SGX paging mechanism (c.f. Section 4.5).

Due to the API of ZooKeeper which is close to a key-value store or a file system API and its limited amount of server-side data processing, ZooKeeper is much more suitable for securing it by extracting parts of its application logic and offloading it to a TEE instead. Hence, the above described TBB approach is applied for SecureKeeper and critical parts of its application logic are secured with trusted execution while most of the service still runs unchanged in an untrusted Java-based execution environment. Thereby, the trusted components are implemented in native code instead of Java in order to avoid the need for a JVM inside the TEE due to the above negative aspects. Consequently, using the TBB approach for securing ZooKeeper with SGX increases security by a reduction of

[5] `https://openjdk.java.net/`

the TCB and the complexity in the enclave. At the same time it provides a lean enclave memory footprint which improves the overall performance of the secure ZooKeeper variant with respect to the characteristics of the SGX technology.

The architecture of SecureKeeper comprises the Java-based ZooKeeper service running mostly unchanged on top of a regular JVM in an untrusted environment. For SecureKeeper small trusted components similar to the ones of the previously described ZPP are integrated into the original ZooKeeper application. Those are responsible for all sensitive data processing in ZooKeeper and are the only entities with the ability to access the cryptographic keys and to decrypt the sensitive user data. The trusted components are implemented in native code executed inside SGX secure enclaves, and are integrated into the Java environment of ZooKeeper by using JNI.

In addition to the generic attacker model defined earlier for this thesis (c.f. Section 3.2.2), SGX provides additional security measures that allow defending against an even stronger attacker. Hence, a typical attacker and trust model for SGX applications is assumed here, that allows strong attackers even with physical access to the target machine and excludes the OS and other privileged software components from the TCB. The attacker's goal is to break the confidentiality and integrity of the sensitive data stored in ZooKeeper and processed in clear text in the TEE. Availability threats are typically excluded in SGX scenarios as the untrusted OS has always the power to not schedule the enclave. In addition, the cloud provider owning the hardware could always disconnect the network or power off the machines.

The resulting TCB of SecureKeeper is relatively small as it neither comprises an OS, a JVM nor most of the ZooKeeper application itself. This is opposed to a set of existing approaches that target the execution of complete applications in a TEE by providing the necessary system support in there. For example Haven [15], SCONE [10] and Graphene [115] provide trusted platforms inside an SGX enclave for execution of complete and unchanged legacy applications. The most notable advantage of the TBB approach of SecureKeeper compared to those approaches is the significant reduction of the TCB. However, the porting overhead is expected to be larger than running a complete application in a TEE assuming a platform like Haven is already available.

Design of SecureKeeper

Figure 4.10 shows an overview of the architecture of SecureKeeper with secure enclaves embedded into the original Java application. As can be seen, each replica contains an individual "Entry Enclave" for each connected client. In addition, there is a "Counter Enclave" embedded into the request processing pipeline of ZooKeeper which is avail-

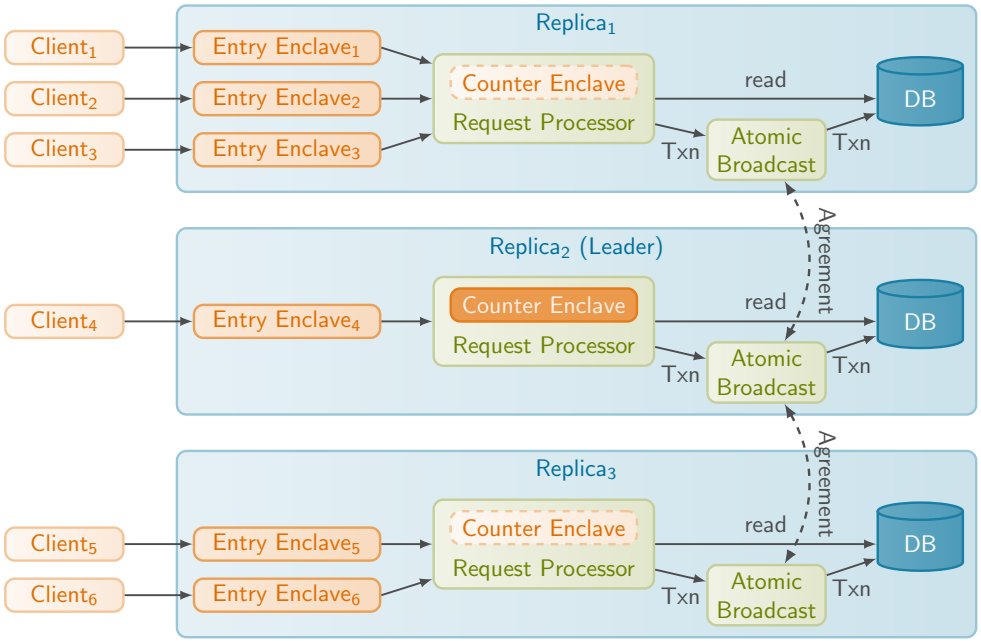


Figure 4.10: SecureKeeper coordination service architecture.

able on each replica but only active on the leader replica as illustrated in the figure. Despite the two types of enclaves integrated into it, SecureKeeper still offers the same fault tolerance properties as ZooKeeper.

The purpose of the **Entry Enclaves** is to accept a TLS connection from a client and terminate the TLS encryption. Next, the entry enclave will encrypt the sensitive data of each passing message with a secret key shared between all enclaves and forward the message to the outside untrusted ZooKeeper request processor component. Afterwards, the message processing of SecureKeeper works exactly like ZooKeeper with the only difference that it now works on encrypted data not the clear text of the data.

In case of requests to SecureKeeper that alter the sequence number counter of a znode the additional **Counter Enclave** is required to execute the server side processing required for sequence numbers that can not work with cipher text as opposed to other ZooKeeper operations. In that case, the untrusted request processor knows when the Counter Enclave is required and forwards the request to it during normal request processing. The Counter Enclave then decrypts the message, calculates and appends the new sequence number and finally encrypts the data before normal request processing continues outside of the TEE by the untrusted request processor.

In the outlined SecureKeeper system, plain text of sensitive data is only available inside the Entry and Counter Enclaves. All other components are only available to access

the cipher text of that data and can work without knowing the plain text. Hence, the untrusted parts of SecureKeeper are Java-based and identical to the original ZooKeeper implementation. Only the enclaves are integrated via JNI into the replicas and implemented in native code.

By using JNI the enclaves can be integrated into the Java-based ZooKeeper application even though they are implemented in native code. For this purpose the data exchanged at the boundary between Java and native code needs to be translated from Java objects to buffers that the C application can use and back, which is done with Java Development Kit (JDK)-provided libraries. For this purpose, we implemented a special shared library called via JNI, that is responsible for launching the enclave and managing its life cycle. In addition, this library makes use of the Intel SGX SDK libraries and handles all interaction of the Java application via JNI with the SGX enclaves.

Request Processing

All sensitive data processing of SecureKeeper is done inside the lean native code enclaves, while most other actions (such as persisting data on disk) are executed by untrusted Java code that makes up a large fraction of the overall code base. Thereby, the general idea of request processing relies on two different kinds of encryption techniques: Between the clients and SecureKeeper (strictly speaking the Entry Enclaves) we implement a *transport encryption* for complete messages using TLS, while we incorporate *storage encryption* between the Entry Enclaves and the untrusted ZooKeeper data store. While the TLS connections between clients and SecureKeeper work with individually negotiated session keys, the storage encryption is based on a secret key shared between all enclaves which is never exposed to the clients.

The mapping of client connections to enclave IDs is done by SecureKeeper in Java using a lookup-optimised data structure based on a hash table. This task can safely be done in the untrusted environment as it does not add an additional risk with respect to the defined attacker model. Each enclave always has to expect wrong or invalid inputs as it would be the case if a message is provided to the wrong enclave. Only the correct enclave will know the used secret key for encryption and be able to decrypt and access the sensitive data. Also only a correct enclave has access to the connection-specific TLS session key and is thus able to create an authentic connection with the client.

Request processing is mostly done inside Entry Enclaves and comprises three crucial steps: Firstly, the transport encryption (TLS) is removed. Secondly, the plain text message is analysed, its type is determined and sensitive fields are identified. Finally, sensitive data of the message is encrypted with the storage encryption key and the al-

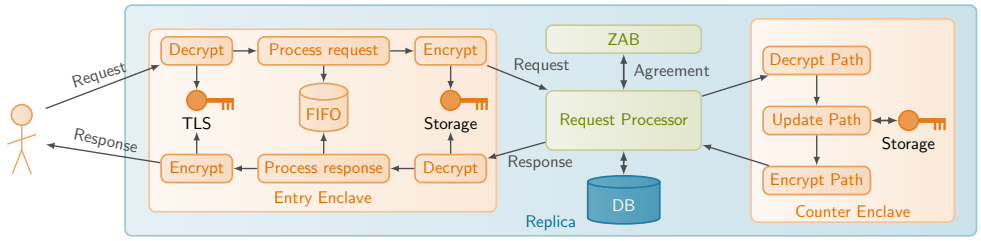


Figure 4.11: SecureKeeper coordination service enclaved request processing.

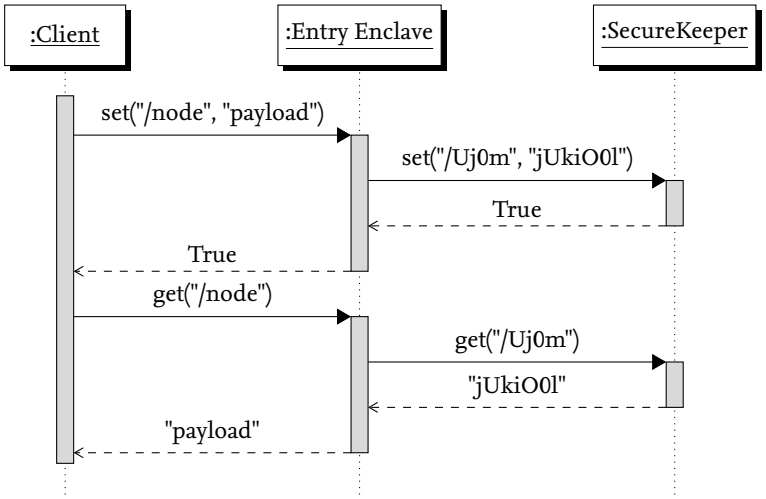


Figure 4.12: SecureKeeper exemplary request processing.

tered message is forwarded outside the enclave to the untrusted request processing component. Responses from ZooKeeper that go through the Entry Enclaves as well before being delivered to the respective clients are handled similarly and pass through the same steps in reverse order. This process is also illustrated in Figure 4.11.

Note that especially the second step of message processing in Entry Enclaves is highly dependent on the message or operation type. A `get()` request contains the znode's path in question, while the according response message comprises the znode's payload field. In contrast, a `set()` request includes the znode's path and payload, whereas the response message to this request contains only the meta data about whether the operation was successful or not. Figure 4.12 provides an illustration of this process at the example of a `set()` followed by a `get()` operation.

Special handling is required on the leader replica for `create()` operations with the sequential flag set. Only in that case, the Counter Enclave is called from the request processing on the leader replica to incorporate the sequence number into the returned znode's name. Due to its low resource footprint, the Counter Enclave is initialised on

all replicas but only active on the leader replica as this is the only place where sequence number changes can happen.

ZooKeeper response messages unlike request messages do not contain a field depicting their message type. As this is crucial for understanding the semantics of the message, this information has to be determined independently in order to correctly handle the messages in the Entry Enclaves. For this purpose each entry enclave maintains a FIFO queue data structure and appends the type of each handled request to it. Exploiting the ZooKeeper's client request FIFO ordering guarantee, this data structure can be used to determine the type of each incoming response message from the untrusted ZooKeeper replica that has to be processed for delivery to the respective client.

Security & Privacy

The znode payload encryption primarily targets the confidentiality protection of the data. However, by application of the Galois/Counter Mode (GCM) mode of the Advanced Encryption Standard (AES) encryption data also gets integrity-protected, as in this cipher mode of AES an additional 128 bit data authentication tag is produced which is stored along with the cipher text for later validation. This adds integrity protection to the znode's payload as it enables the detection of any modifications to the encrypted data. As this procedure increases the length of a znode's payload the respective meta data of the znode has to be updated as well.

Path encryption of znodes in SecureKeeper is based on the individual chunks of a znode's path delimited by the slash character ("/"). Each chunk of a path is handled and encrypted individually and based on the rationale of the payload encryption. However, as an Initialization Vector (IV) for the encryption a hash of the complete path from the start to the chunk in question is used. Inherently this ensures that an IV is never used twice as ZooKeeper does not allow existence of two znodes with the same name.

A special case of path encryption is the `ls()` operation of ZooKeeper. In this case, the request contains the path of a particular znode, but the entry enclave must be able to decrypt the paths of all subsidiary znodes of the given one. In order to enable this special case, the IV used for encryption of a path chunk is appended to its cipher text.

It is important to cryptographically connect the znode payload to its path in order to prevent an attacker from exchanging the payload of a znode with the one of a different znode. This is dangerous even when the attacker is not able to decrypt the path or payloads as it could affect the assignment of a user to a different user group with higher privileges in a ZooKeeper-controlled distributed system for example. In order to prevent such kinds of attacks, a hash of the znode's path is included in the znode's

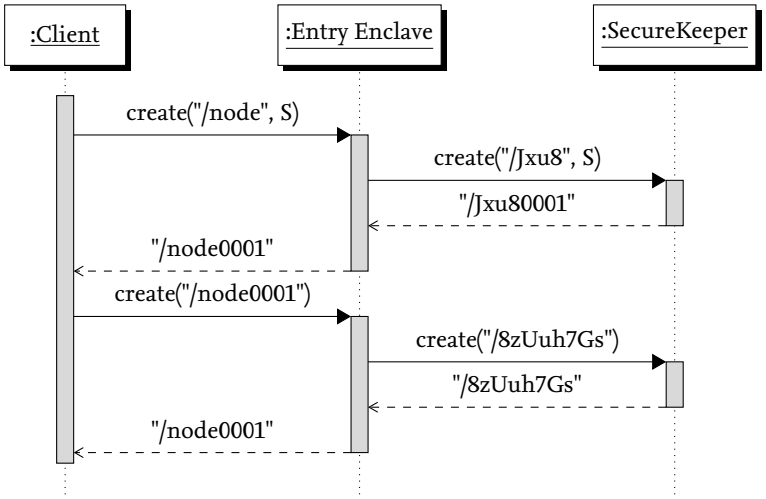


Figure 4.13: SecureKeeper sequential znode creation name clash.

payload before encryption. The entry enclave ensures that data is only returned to the users after successful validation of this constraint.

During the creation of a znode with the sequential flag, ZooKeeper will compute a sequence number and append it to the znode's name before returning the path of the created znode back to the client. With the request processing of SecureKeeper as described so far, the untrusted ZooKeeper request processing would append a sequence number to the path received by the entry enclave which is already encrypted at that time. While this behaviour would not cause crashes, it could lead to conflicts in certain cases: assuming a user creates a znode `/node` with the sequential flag set, SecureKeeper would add the sequence number and return `/node0001`. If in that case, the user afterwards creates a znode `/node0001`, this path would get completely encrypted by the entry enclave (including the sequence number), and thus, would *not* clash with the already existing znode with the same name. Figure 4.13 illustrates this conflict; note, that the entry enclave in this example is assumed to remember that the znode was created with the sequential flag and only tries to decrypt the front part of the path. In both cases the returned znode path to the client is `/node0001` while on the server side two nodes have been created.

In order to solve the above described possible name clash during creation of sequential nodes, SecureKeeper introduces a second enclave. The *Counter Enclave* is integrated into the request processing of SecureKeeper and only active on the leader replica, because it is only needed for `create()` requests which are writing requests that are all forwarded to be processed by the leader replica. Hence, the Counter Enclave is called from the request processing of SecureKeeper in order to correctly encrypt the sequence

number into the final znode path returned to the client. Therefore, it requires access to the same secret key that all other (entry) enclaves also use for encryption.

The sequence number of a sequential node in ZooKeeper is derived from the parent node's meta data, which we keep unencrypted in SecureKeeper as well because of several reasons. Only an attacker on the same machine could retrieve the current sequence number from the meta data fields, however, such an attacker could also derive the sequence number solely from passively observing the activity on the replica even if everything is encrypted. In addition to that, only the sequence number counter itself is maintained unencrypted, however, the full path name of a sequential node in the SecureKeeper data base including the appended sequence number is still encrypted.

Deployment and Key Management

As described above each enclave in SecureKeeper requires a secret key for encryption of the sensitive data which must be provisioned to the enclaves in a secure way. The general idea of achieving this is to provide the key to the enclaves only after successful remote attestation of the enclave by an administrator. In order to allow automatic restarts of an enclave and to start new instances of the same enclave without the need to attest them first and deploy the key, the enclaves could persist that key securely outside the enclave using the SGX sealing mechanism. By this, an administrator has to bootstrap the SecureKeeper cluster and perform remote attestation once only, but the SecureKeeper cluster is able to restart or instantiate new enclave instances automatically afterwards. This is an important feature, as otherwise fault tolerance and scaling to current demand is not easily possible without manual administrator intervention. Thereby, SGX sealing protects the secret key even when persisted to disk and only allows a trusted enclave instance to access the sealed data.

In addition to the above key management approach, the clients require a procedure to determine whether they could trust a SecureKeeper replica or not. Essentially this implies the same criteria as remote attestation. However, requiring the execution of a full remote attestation procedure by each client for each connection to SecureKeeper would lead to an enormous complexity of using SecureKeeper. In addition, the remote attestation would increase request latency by additional network messages quite significantly, rendering the approach practically infeasible. Hence, in order to guarantee that SecureKeeper clients only interact with trustworthy enclave instances, the rationale of SecureKeeper is to establish authenticity via TLS. Only after successful remote attestation of an enclave by an administrator, the TLS private key is injected into the enclaves. By this, a client that is able to establish a valid TLS connection with a correct

server certificate implicitly knows that the enclave has been successfully attested by an administrator because otherwise the TLS private key would be unknown to the enclave.

Transport encryption in SecureKeeper is implemented as AES encryption using the SGX SDK-provided encryption library without a full TLS cipher negotiation. Choosing this cipher is reasonable from a security perspective and especially in an SGX setting to achieve good performance as the SDK library is able to accelerate AES encryption in hardware, which is common practice for non-SGX applications as well. We do not expect a significant distortion of the performance measurements of SecureKeeper due to the missing cipher negotiation, as this only impacts the connection establishment and we assume long running client connections.

Evaluation

This section describes the evaluation of SecureKeeper in order to quantify the achieved reduction of TCB and the achieved performance of the secure ZooKeeper variant SecureKeeper. Thus, the evaluation comprises measurements of the TCB and the memory footprint of the SecureKeeper application and its enclaves. In addition, the performance of SecureKeeper is evaluated and compared with the insecure ZooKeeper in order to measure the performance impact of usage of SGX.

The evaluation of SecureKeeper was done on a cluster of four SGX-capable machines equipped with a Intel Core i7-6700 processor, 24 GB RAM, 256 GB SSD storage and 4× Gigabit Ethernet. Three machines were used for running the SecureKeeper replicas, while three is the minimal number of replicas for a ZooKeeper cluster. The fourth machine has been used to simulate the clients.

As a baseline for comparison, ZooKeeper with TLS-protected client connections is used. This variant of ZooKeeper by the original ZooKeeper developers was publicly available as an alpha version at the time of developing SecureKeeper and already included TLS encryption support of connections between clients and ZooKeeper replicas implemented in Java. In case of SecureKeeper the main difference is that the transport encryption endpoint does not reside in Java which is untrusted in that case, but instead the transport encryption is decrypted in native code inside the enclaves.

Table 4.2 illustrates the size of the original ZooKeeper server code base compared to SecureKeeper with an overall TCB of about 4,000 SLOC. As can be seen, the assumption that the security-critical parts of the application logic are only a small portion of the complete code base was correct and the partitioning of ZooKeeper led to a TCB much smaller than the ZooKeeper code base. More than 60% of this trusted code is responsible for message (de)serialization and borrowed from the original ZooKeeper C

	Component	Language	SLOC
trusted	(De-)Serialization	C	2514
	Counter and entry enclave	C	985
	SDK-generated code	C	572
	Total trusted		**4071**
untrusted	ZooKeeper Server	Java	33851
	Enclave interfaces	Java	154
	Enclave management	C	296
	SDK-generated code	C	262
	Total untrusted		**34563**
	Total SecureKeeper		**4783**
	Total		**38634**

Table 4.2: Size of code base of SecureKeeper components.

bindings. The remainder of the TCB implements parameter passing and glue code for the message flow through the enclaves. The total amount of trusted code of SecureKeeper accounts for roughly 12% of the complete ZooKeeper code base.

Apart from the trusted code, developing SecureKeeper also required adding new untrusted code to the final application as shown in Table 4.2. This comprises creation, initialization and maintenance of the enclaves, parameter passing across the enclave boundary and the JNI interface. Only three lines of the original ZooKeeper code base had to be changed in order to integrate the new classes for SecureKeeper, all others have been added, thus, we call our approach minimally invasive to ZooKeeper.

The above TCB breakdown focuses on the actual application logic code and does not account for inevitable library and runtime dependencies of any Java or C application. On the one hand for example, in order to build an SGX enclave, a set of Intel SGX SDK libraries is required that comprises about 18,000 SLOC including a cryptographic library. Furthermore, a reasonably lean full-featured TLS library such as mbedTLS consists of about 55,000 SLOC, however, this includes a large number of ciphers and functionality that is not all required at the same time, but usually used only partly dependent from the cipher negotiation. On the other hand, in order to run any Java-based application, the Java runtime is required, which comprises more than 6,000,000 SLOC usually running on top of an OS with more than 20,000,000 SLOC (the Linux Kernel consists of 24,000,000 SLOC). Certainly, excluding the OS and Java runtime from the TCB improves the security level significantly, however, the potential to additionally save SLOC from the TCB by tailoring libraries used inside the enclaves remains.

Next we evaluate the performance degradation due to the stronger security guarantees of SecureKeeper and compare it to the performance of ZooKeeper. As a preliminary experiment we determined the number of parallel client threads that achieve the high-

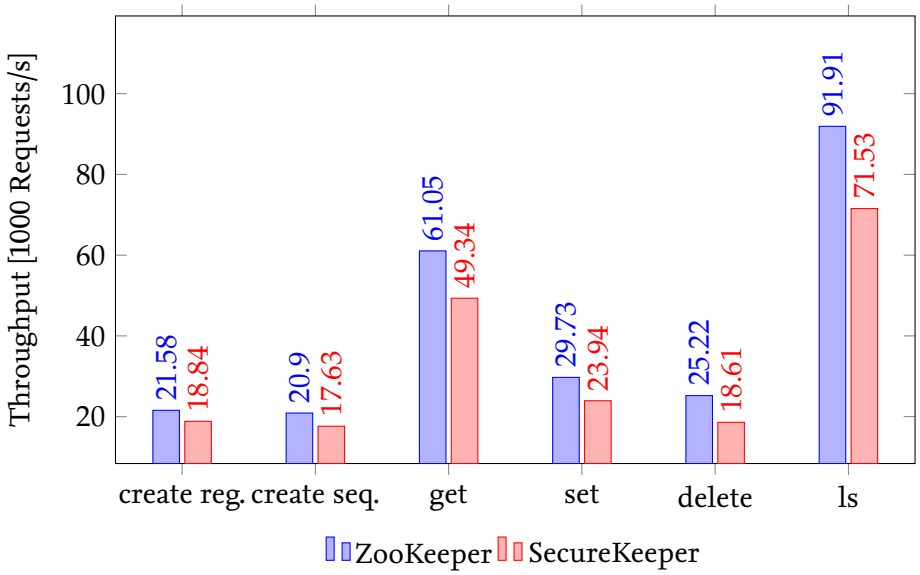


Figure 4.14: SecureKeeper throughput of synchronous operations.

est throughput with an unchanged ZooKeeper server by incrementally increasing the number of simultaneous clients connected to the cluster issuing requests. For this experiment we chose a realistic workload of a 70:30 ratio of `get()` and `set()` requests and reached the peak throughput at 300 client connections for synchronous requests. For asynchronous requests we measured the highest performance for 5 client threads, each with a maximum of 200 concurrently pending requests—1000 requests simultaneously in flight in total. For our throughput measurements we compare SecureKeeper against ZooKeeper with clients connected via a TLS connection as a baseline. Thereby, the clients are distributed equally across all available SecureKeeper or ZooKeeper replicas and the requests contain a realistic dummy payload of up to 4096 random Bytes.

In our experiments, we evaluated the `create()`, `get()`, `set()`, `delete()`, and `ls()` (or `getChildren()`) operations. In case of creates we further distinguish creation of regular znodes and sequential znodes, the latter requiring an additional enclave entry into the Counter Enclave. The results of our measurements are illustrated in Figure 4.14 and Figure 4.15. As can be seen, with a performance overhead of about 22.4% (20.2% for synchronous operations and 24.6% for asynchronous operations), the performance of SecureKeeper is close to ZooKeeper. For most operations the overhead of SecureKeeper is low, however, the `ls()` operation has relatively high overhead due to the path encryption approach that requires additional computation inside the enclaves.

The fault tolerance properties of SecureKeeper correspond with the ones of ZooKeeper as the enclaves are directly integrated into the server application and can all tolerate ar-

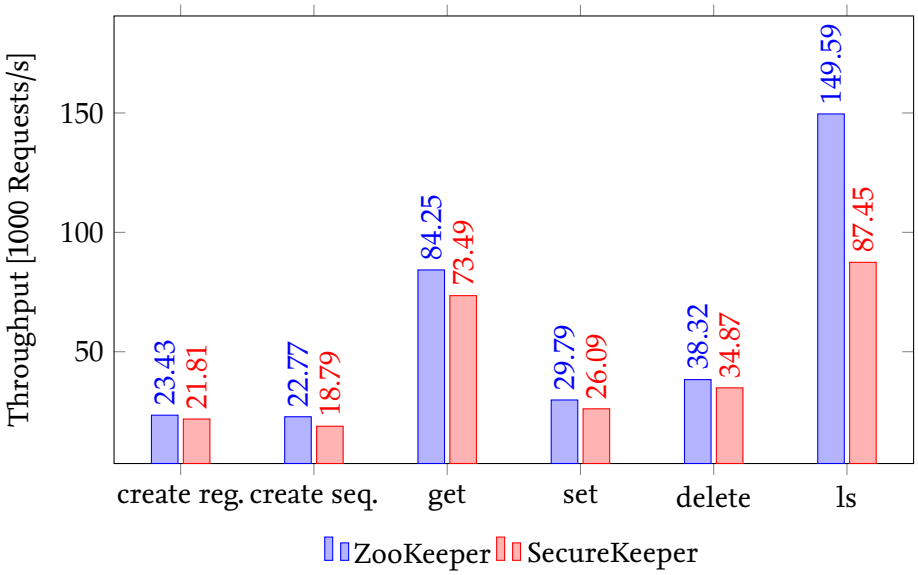


Figure 4.15: SecureKeeper throughput of asynchronous operations.

bitrary crashes. From a client's perspective, a replica that does not respond in time is considered faulty and the client replaces the connection to the cluster by a new connection to another replica. A failure of the SecureKeeper leader replica leads to a leader election in the same way as with a regular ZooKeeper cluster.

SecureKeeper Conclusion

In conclusion, SecureKeeper can be seen as a replacement of ZooKeeper with the same consistency guarantees and properties. However, in addition to ZooKeeper, SecureKeeper offers a much stronger security level than ZooKeeper by protection of the confidentiality of all data stored inside the data base with secure SGX enclaves. Furthermore, SecureKeeper also offers basic protection of integrity of stored user data.

The development of SecureKeeper has demonstrated the required actions to manually partition a real-world application for usage of trusted execution in an untrusted cloud setting. Hence, it provides a proof of concept of the feasibility of partitioning a complex legacy application according to the TBB approach with SGX.

4.7 Trusted Execution Metrics and Design Criteria

In order to generalise the above application partitioning approach and emphasise the crucial parameters to consider while partitioning a legacy application, we investigate

and partition the Voldemort key-value store[6].

Voldemort is another representative of a Java-based data-handling service, is used at LinkedIn and resembles an open source clone of Amazon's Dynamo store [31]. It supports distributed setups with sharding of the stored data across multiple hosts and features a horizontal scalability of both, read and write accesses. However, Voldemort is not a relational database system, as to its clients it offers a relatively simple get/put API. According to its authors it is a "big, distributed, persistent, fault-tolerant hash table"[7].

4.7.1 Performance Metrics

The performance of an SGX application is highly dependent from the memory footprint of the enclave due to the high SGX paging overhead. Strictly speaking the *working set* memory footprint of an enclave is crucial. This means that only the total amount of "frequently used" pages of an enclave is important in this regard, as less frequently used pages can be evicted and do not occupy precious EPC space. Thus, keeping the working set memory footprint of an enclave small—and especially below the available EPC size—is important for the performance of the enclave.

The memory footprint of an enclave is defined by multiple factors, such as the compiled enclave binary containing the source code of the enclave, but also additional pages allocated during enclave creation for usage as stack, heap or SGX-internal data structures such as thread contexts. Therefore, many factors have to be taken into account in order to reduce an enclave's memory footprint. However, it is also significant to consider the memory access pattern of an enclave in order to allow prefetching technologies to work most efficiently. For example sequential memory accesses could be optimised by a prefetcher much more easily than random accesses.

In addition to that, the number of execution mode switches between the untrusted and trusted environment is a crucial performance factor of SGX applications [121, 40], but similarly this is also crucial in TrustZone applications. Both trusted execution technologies require security-critical actions during execution mode change, that cause delays compared to normal execution. For example CPU register contents have to be saved, restored or even wiped and additional permission checks are required in order to prevent illegal accesses or leakage of sensitive data.

In order to respect this property, it is substantial to carefully select which part of an application's source code is offloaded to an enclave. In some cases it could be even advantageous to move a small function inside an enclave, even though the function does

[6] Voldemort `http://www.project-voldemort.com/voldemort/`
[7] `http://www.project-voldemort.com/voldemort/`

not process any sensitive data, in order to prevent frequent enclave exits. Even copies of the same function inside an enclave and outside could be useful under this premise.

Finally, not all enclave entries or exits are completely under the control of the developer of the enclave. Interrupts for a CPU that is currently in enclave mode cause an AEX of that CPU, that forces that CPU to save the active execution context and leave the enclave and continue execution of the Interrupt Service Routine (ISR). Only after finishing ISR execution, the CPU may enter the enclave again and continue execution at the previously stored location.

Interrupts—and CPU exceptions, that constitute another cause of an AEX—are certainly not subject to omission. However, it could be possible to pin enclave threads to CPUs and configure interrupts in a smart way, such that interrupt handling and enclave execution is separated to different CPU cores. This would require fine-grained adjustments during runtime dependent on the current interrupt and system load situation, and has not been further investigated in this thesis. However, Intel itself has investigated this aspect for SGX applications in the form of so called "switchless calls" as part of the Intel SGX SDK. Also several research works have investigated the cost of execution mode changes and tried to improve their efficiency [121, 85, 112, 40].

4.7.2 Security Metrics

As studies have shown and described in Section 3.1.1, the TCB of an application constitutes one of its most important security metrics, as the amount of source code correlates with the number of bugs it contains. Thus, the amount of code executed inside an enclave is considered security-critical. However, the TCB of an enclave not only consists of the actual business logic of the application, but other components described in the following further increase the overall TCB.

In order to enter and exit an enclave, certain tasks have to be done, such as handling CPU register contents and switching between the untrusted and trusted stack for example. These actions are usually not explicitly done by the enclave developer, instead for this purpose the Intel SGX SDK generates code that is linked to the untrusted and trusted objects. Furthermore, the Intel SGX SDK also comprises special libraries that are supposed to be used inside an enclave and add up to the TCB such as the Intel SGX SDK's special libc and cryptographic library. These libraries are modified in order to be able to run inside an enclave under the given limitations of enclaves such as the lack of the ability to execute system calls and certain CPU instructions for example.

In addition to an enclave's TCB, the interface of an enclave to the outside is also a security-critical metric. Due to the attacker model of SGX comprising an untrusted

OS, the enclave can never trust any inputs from the OS. Hence, it is the enclave's responsibility to check sanity of all provided inputs, be it return values of calls by the enclave to the OS (c.f. IAGO attacks [15]) or arguments of calls from the outside into the enclave. Therefore, the number and signatures of enclave calls (ecalls) is considered security-critical. These requires checks also lead to an increase of the enclave's TCB.

Another factor affecting the overall security level of an SGX application is fault isolation. In certain cases it might be beneficial to split the application logic up into multiple enclaves. In this case, bugs in the application logic of one enclave do not necessarily also affect other enclaves. Furthermore, instantiating the same trusted code multiple times in separate enclaves could be beneficial as well. For example, if each client connection of a server application is handled by an individual enclave, a bug in the application logic could only affect the data of a single user and could additionally only be exploitable after authentication, which excludes unauthenticated attackers from the attack surface. Hence, instantiating enclaves multiple times and splitting up application logic into isolated enclaves could further increase security and limit the harm of exploitable security vulnerabilities in the enclave's code. Finally, this approach also increases security by implementing isolation by means of hardware instead of isolating inside an enclave solely by means of software.

4.7.3 Secure Voldemort Key-Value Store—SUE MUE Enclaves

According to the above defined security and performance metrics of SGX applications, we partitioned the Voldemort key-value store application in two different designs. The first design called *Single User Enclave (SUE)* is optimised for security and minimises the TCB of the enclave, while the second design aims for good performance and is called *Multi User Enclave (MUE)*. In this definition, the enclave design of SecureKeeper resembles a SUE, therefore, we retrofitted the MUE design for SecureKeeper and evaluate the security and performance of both enclave designs for both these use case applications.

In case of our two use case applications (ZooKeeper and Voldemort key-value store), the SUE enclave design is connection-agnostic, in the sense that connection management and assignment of messages to the right enclave is done in untrusted code outside the enclave. The enclave itself has no notion of connection IDs and only stores one encryption key that will inherently only work with the right connection as this key is negotiated individually for each connection. Hence, the untrusted code has the ability to forward messages to the wrong enclave, but confidentiality is not compromised by this as the wrong enclave is not able to decrypt the data and this attack vector is an availability threat. This allows saving code and data required for connection management

in the enclave and reduce its complexity and TCB as well as its memory footprint.

In contrast to this, the MUE enclave design includes connection management inside the enclave and supports handling of multiple connections by the same enclave. This requires additional code (for example a hash table implementation) and memory for storing meta data of each connection such as session keys, and also increases the compute time inside an enclave, for example for connection context lookup by ID.

Even though a single SUE is smaller, the overall memory consumption of the MUE design is lower, as the enclave is held in memory only once and handles multiple connections. Hence, the MUE design allows reusing the enclave code containing all libraries for all client connections which is not possible otherwise as enclaves are not able to share memory pages. Therefore, we expect the MUE design to perform better than SUE enclaves in an environment with high numbers of clients such as the cloud.

However, the MUE design implies higher contention of thread synchronisation primitives, as processing multiple connections inside the same enclave requires multiple threads entering that same enclave and coordination of access to shared data structures. While Intel allows entering the same enclave simultaneously by multiple CPUs and parallel execution inside an enclave, in order to control parallel access to shared data structures synchronisation primitives like mutexes are required. The implementation of mutexes in the Intel SGX SDK already takes into account the costly enclave exits that would be required to execute the mutex-related system calls. Therefore, the implementation by Intel tries to wait for the mutex inside the enclave using spin locks for a short period of time, aiming at preventing the costly exit, and only exits the enclave when the waiting time exceeds a threshold.

In summary, the SUE design with a minimal TCB optimises for security. In contrast, the MUE design works with an increased TCB and connection management inside the enclave and considers performance criteria and SGX-specific limitations in order to increase the performance at the cost of security. Therefore, an enclave designer must always face a trade off between performance for security.

4.7.4 Evaluation

In order to compare the two proposed enclave designs SUE and MUE regarding the security and performance metrics we introduced above, we have measured their TCB, working set memory footprint and request throughput.

Table 4.3 illustrates the TCB of the two enclave design approaches for SecureKeeper and the secure Voldemort key-value store—called *Dumbledore* from now on. When comparing the amount of code required inside the enclave with the amount of code of the

	SecureKeeper	Dumbledore
TCB	4.2 (1.7)	0.7
Untrusted Code	34.5	76.3
TCB Ratio [%]	12.1 (5.1)	0.9

Table 4.3: Enclave TCB in thousands of SLOC.

	SecureKeeper	Dumbledore
Start pages	322 (1.26 MB)	295 (1.15 MB)
Warm pages	94 (0.37 MB)	67 (0.26 MB)
Enclave binary size	1.2 MB	1.2 MB

Table 4.4: Enclave EPC memory footprint.

original application before partitioning, it can be seen that the TBB approach significantly reduces the TCB for both use cases. In case of SecureKeeper, the enclave's code base consists of 4.2k SLOC, which is 12.1% of the original 34.5k SLOC of ZooKeeper. Hereby, a large fraction of the TCB is solely responsible for (de)serialisation of network messages. If this code is set aside, the remaining TCB accounts for 1.7k SLOC or 5.1% of the ZooKeeper code base (numbers illustrated in parentheses in Table 4.3). In case of Dumbledore the TCB of 0.7k SLOC of the 76.3k SLOC of Voldemort constitute only 0.9%. Hence, this provably shows that the potential for reducing the TCB by application partitioning with the TBB approach is significant. Note that, as Dumbledore is an SGX application like SecureKeeper the TCB also needs to comprise the Intel SGX SDK libraries amongst others (c.f. SecureKeeper evaluation in Section 4.6.3).

In order to measure the working set memory footprint of an enclave, we used *sgx-perf* [120], that allows to determine which pages of an enclave are actually accessed during enclave execution. The rationale of sgx-perf is to remove access permissions to all enclave pages in the page table and count the occurring page faults caused by the missing permissions when the enclave tries to access that memory. By this, the enclave's working set memory footprint can be measured at page-granularity.

In general, an enclave consists of several different memory regions. During enclave creation, the compiled binary of the enclave code is written to the ELRANGE. Additionally, the creation process creates pages for storage of the enclave's heap, a stack region for each thread and other thread-specific data structures.

Table 4.4 shows the results of this evaluation at the example of the MUE enclave: after initialisation the enclave's working set memory footprint constitutes more than

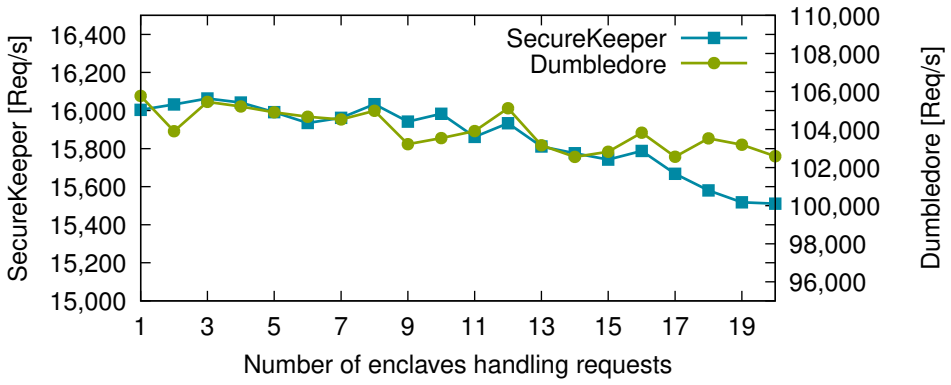


Figure 4.16: SecureKeeper and Dumbledore performance.

1 MB for both use case applications. Once the benchmark has warmed up, the memory consumption drops to only 94 pages for SecureKeeper and 67 pages for Dumbledore respectively. "Warmed up" in this case means the enclave initialisation process is finished and the enclave is processing actual requests. In both cases heavy SGX paging is avoided with a memory consumption far below the targeted EPC limit. The SUE enclave's memory footprint resembles those measurements, however, for each new client a new SUE enclave would be required, while we measured only 5% more memory consumption for each new client with our MUE enclave. This demonstrates the expected higher memory efficiency of the MUE approach compared with SUE.

In Figure 4.16 we show the performance measurements of our two use case applications SecureKeeper and Dumbledore both with the MUE enclave. For the experiments we used server-grade machines with 24 Intel E5-2430 v2 cores as clients to provide sufficient load for the SecureKeeper and Dumbledore cluster. The client machines ran a special custom evaluation tool by us, that allowed fine-grained evaluation parameter tuning for configuring and coordinating the required client threads issuing requests from several machines at the same time to the respective cluster being evaluated. All experiment results constitute average values of multiple runs, each with their own warm up phase and a freshly started target cluster. The SecureKeeper and Dumbledore cluster has each been executed (consecutively) on the same SGX-capable E3-1230 v5 server machines with SGX in hardware mode and an EPC size of 128 MB. For this measurement a fixed number of MUE enclaves handles the arriving requests of 72 clients spread

equally across all enclaves. Hence, for $x = 1$ a single MUE enclave handles all clients alone, while the SUE approach equals $x = 72$, which is not shown in the figure as we could only measure up to 20 enclaves for stability reasons. However, the measurement already clearly shows that the performance of a single enclave handling all requests is better: for both applications we measured a performance drop of about 3% in the range of one up to 20 enclaves already. We explain this by a more efficient CPU cache usage with lesser enclaves, as with a high number of enclaves more often cache lines of other enclaves have to be evicted. We expect that for more enclaves the performance would decrease even more, and eventually hit the EPC threshold were performance drops significantly due to SGX paging.

In summary, the above evaluation proves that the TBB approach could be transferred with reasonable effort to other comparable applications. The TCB after partitioning is relatively small for both use case applications compared with their original full code base. This is true for both the SUE and MUE enclave design, while the MUE requires a small amount (less than 100 SLOC) of additional SLOC in order to implement the support for multiple connections in the same enclave. Memory consumption is far below the EPC size which prevents heavy SGX paging with significant performance impact, however, the MUE approach is much more efficient in this regard for high numbers of users. Due to more efficient CPU cache usage, the MUE approach also shows higher throughput compared to the SUE enclave design. Hence, it can be concluded, that the SUE indeed provides the highest level of security, and must be traded off against the MUE approach which is more practical with better scalability, higher performance and more efficient EPC memory usage.

4.8 Related Work: Trusted Application Components

Before widespread availability of trusted execution techonologies in commodity hardware components, computation on sensitive data in an untrusted environment could be done for example using homomorphic encryption schemes [64] or keyword search over encrypted data [92, 91]. However, the biggest disadvantage of homomorphic encryption schemes, despite the fact that general purpose computation by usage of Fully Homomorphic Encryption (FHE) is possible by now, is their performance which is still too low to be practically usable [12, 73].

Some other approaches offload sensitive computation to separate trusted (hardware) components like a secure coprocessor. For example, CheapBFT [57] uses a trusted monotonic counter to relax the resource requirements for their Byzantine agreement protocol which is implemented by a Field Programmable Gate Array (FPGA) device.

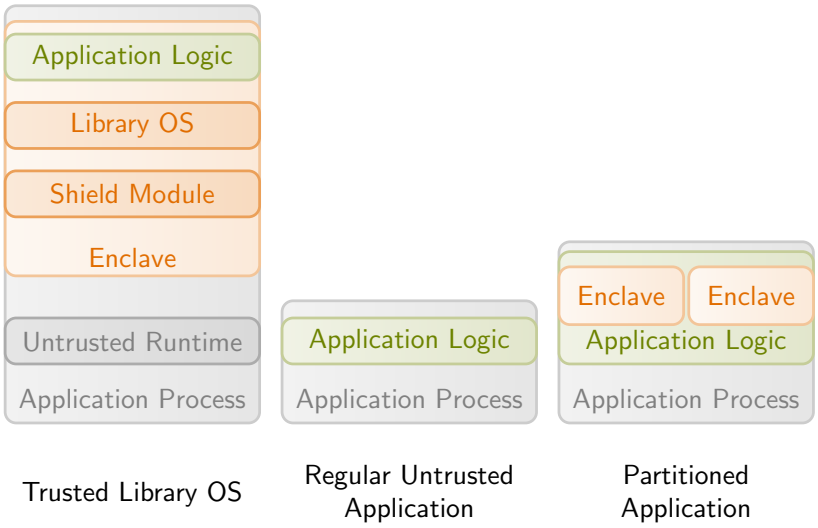


Figure 4.17: Comparison of Enclave Application Approaches.

TrustedDB [12] proposes a trusted relational database engine by integration of an additional cryptographic coprocessor that participates in query processing. Both of these approaches would not satisfy the requirements of a commercial public cloud with tens or hundreds of customers co-located on the same hardware platform each with their individual software components in the TEE. Also, one of the most attractive properties of TrustZone or SGX technology is their widespread availability in consumer CPUs as opposed to highly specific tamper-proof hardware components such as the cryptographic coprocessor used in TrustedDB [12].

The advent of trusted execution in commodity hardware created new opportunities. Initially, several research projects investigated the deployment and execution of unmodified applications inside SGX enclaves without significant porting efforts [15, 115, 10, 106]. The advantages of such approaches comprise a huge reduction of porting efforts compared to application partitioning and the theoretical possibility to secure even closed source applications. However, the TCB overhead of such platforms is significant as it includes a library OS, for example in case of Haven [15], Graphene-SGX [115], SGXKernel [111] or SGX-LKL[8], or a significant amount of trusted code originating from system interface abstractions inside the enclaves [10, 106]. As can be seen in Figure 4.17, a partitioned application allows a much leaner TCB as no library OS is required in the enclave, and only selected parts of the application logic are trusted. Another aspect is that existing approaches either focus more at minimisation of the interface to

[8]SGX-Linux Kernel Library (LKL) at Github: `https://github.com/lsds/sgx-lkl`

the trusted environment [15, 115] or the overall TCB [106, 10]. Application partitioning constitutes a hybrid solution here by sacrificing the low porting effort and instead aiming at primarily optimising the overall TCB while keeping a lean interface with the outside world and only little system support inside the enclave at the same time.

Schuster et al. [102] have proposed a system that exploits the compartmentalised nature of the MapReduce framework [30] and achieves a relatively low TCB by offloading the `map()` and `reduce()` functions to a secure enclave. While this approach is limited to MapReduce applications, our work with SecureKeeper describes the procedure of extracting sensitive parts of a complex existing application to a secure enclave by partitioning in general. Also, to the best of our knowledge, our work with SecureKeeper [21, 22] (and later Dumbledore [19]) was the first practical paper that described the partitioning process of an existing application for usage with Intel SGX and evaluated it on real hardware. After our publication, several other related works have been published that proposed SGX-secured applications: Pires et al. [90] propose another specific application secured with the SGX technology. Their system constitutes a publish-subscribe system, that aims at allowing deployment of the message exchange node (called "routing engine") in an untrusted cloud setting, therefore porting the routing engine partly to an SGX enclave. Glamdring [67] proposed a source-level partitioning framework, that aims at partitioning applications automatically. While this approach reduces the porting cost, informed manual shifting of boundary between trusted and untrusted code wisely has the potential to achieve better performance and a leaner TCB. STANLite [100] provides an SGX-secured SQLite database, that instead of partitioning a database system consists of a lean SQLite database engine for embedded scenarios that is ported as a whole into an enclave and stores data encrypted outside the enclave. A more complex database is proposed in EnclaveDB [94], which constitutes a relational database ported to run with SGX. However, in contrast to SecureKeeper, the authors assume a much larger EPC memory size and hold all sensitive data such as tables and indexes in enclave memory. ShieldStore [58] is an in-memory key-value store secured with SGX, that stores the data outside the enclave and protects its integrity using a Merkle tree whose root lies in the enclave. Compared to SecureKeeper, ShieldStore also respects the limited EPC memory but achieves stronger integrity protection, however, for this it requires more application logic inside the enclave.

5 Modern Software Architectures in the Context of Trusted Execution

In the previous chapter, partitioning of legacy applications for usage with trusted execution has been covered. This chapter investigates modern software architectures that diverge from the established monolithic software design and split large applications into smaller components that are easier to maintain and have the ability to be developed, tested and deployed independently. The inherent partitioning of software already in its design could open new opportunities and simplify or even avoid the complex application partitioning process as described in Chapter 4. This investigation is executed at the example of the serverless cloud computing paradigm, a modern approach that further abstracts the cloud platform's architecture and is based on our paper "Trust More, Serverless" (SYSTOR'19).

Currently workloads in data centres evolve more and more towards an increasing number of smaller tasks consolidated on the same servers [89]. In this context, the FaaS cloud paradigm aims at a fine granularity and tries to further reduce overhead and offload more tasks to the cloud provider [35]. Especially because the design of SGX was initially targeting securing libraries of applications rather than complete legacy applications, this cloud paradigm is considered suitable not only for trusted execution in general but for usage with Intel SGX in particular. Therefore, in this chapter we introduce our Trusted Serverless Platform (TSP) that allows securing selected functions of FaaS applications with Intel SGX while retaining the key characteristics and especially the benefits of the FaaS paradigm. Thereby, the TSP is presented in two distinct variants in this chapter: the *Secure DukTape Lambda Platform (SeDuk)* based on the lean JavaScript engine Duktape, and the *Secure Google V8 Lambda Platform (SeGoo)* with the high performance Google V8 engine at its heart.

5.1 Function-as-a-Service (FaaS) and the Lambda Model

Initially, the idea of IaaS was to increase resource utilisation and provide ease of scalability to customers, while at the same time taking the risk of high investments off their

Table 5.1: Comparison of existing serverless platforms.

Platform	Supported Language Env.
OpenLambda [47]	Node.js, Python, …
OpenFaaS [84]	Node.js, Python, …
OpenWhisk [7]	Node.js, Python, Java, Swift, Go, Scala, PHP, Ruby
Amazon AWS Lambda [4]	Node.js, Python, Java, Go, .NET Core
Microsoft Azure Functions [80]	Node.js, C#, F#
Google Cloud Functions [42]	Node.js, Python
Cloudflare Workers [27]	JavaScript

shoulders. This idea evolved in PaaS that further shifted the management overhead off the cloud users to the cloud provider and allowed more efficient resource sharing. FaaS, a modern cloud paradigm, builds upon that thought and allows the customers to completely stop thinking about single server machines and their management—thus the frequently used term *serverless*.

FaaS aims at reducing the management overhead for an application developer and offloading more tasks to the cloud provider. Thereby, with FaaS, the application logic is deployed in the cloud on the granularity of standalone functions that can interact with each other and be called from their users. Consequently, FaaS can be seen as the continuation of the intentions of the earlier cloud computing approaches IaaS and PaaS.

As a FaaS pioneer Amazon coined the term "Lambda" for FaaS functions [4], thus, for the sake of better readability, throughout this thesis FaaS functions are simply denoted as *Lambdas*. Lambdas are small inherently stateless functions, usually written in interpreted languages like JavaScript or Python [47]. They are provided by the FaaS application developer, and often executed in isolated environments like Docker containers on the cloud provider's machines [119]. In Table 5.1 we summarise some prominent existing FaaS platforms and their set of supported programming languages for Lambda development. As can be seen, JavaScript (often with Node.js) and Python are the most frequently supported programming languages for Lambdas, and both supported by a majority of popular existing Lambda platforms.

Large Lambda applications are comprised of multiple Lambdas forming a holistic application by interconnecting the Lambdas that can use a common database or other forms of persistent storage to share data. Lambdas can be called directly or triggered due to events, such as storing a file in a data store. Essential to Lambdas is their ability to be started quickly and also to automatically scale across multiple machines according to the current demand.

As mentioned above, Lambdas itself are usually stateless and only active when being actually called. This implies the common payment model for FaaS platforms which charges customers on a per-request basis based on the actually used resources and even allows zero cost for idle Lambdas as those could even be unloaded from memory by the platform. On the other hand, this property requires a Lambda platform to have the ability to quickly load and start a specific Lambda once a request to that Lambda arrives. Hence, the cold start latency of Lambdas is a crucial property. Since Lambda platforms obey the serverless paradigm and users are not involved in server management, the platform decides where to execute a specific Lambda, and in turn, is able to quickly adapt to the current load and automatically scale Lambdas across machines if necessary.

As most Lambdas are written in interpreted languages it is possible to deploy only the Lambda's code in interpreted language on workers and share the interpreter between different Lambdas [47]. This is already a common habit of commercial providers such as Amazon which uses containers for Lambdas and reuses the interpreters for different Lambdas to improve startup latency and overall resource efficiency[1]. Furthermore, Lambdas are supposedly short-lived which allows the provider to reuse a Lambda's container for different Lambdas while not in use. Memory of idle Lambdas can even be freed (or reused) which allows providers to charge "zero cost" for idle Lambdas [47].

5.2 Trusted Serverless Platform (TSP)

This section details the design considerations and describes the architecture of the Trusted Serverless Platform (TSP) created as part of this thesis. We first analyse the requirements of such a platform regarding the user-facing features and the provider-facing capabilities. Then, we discuss crucial aspects like scalability, cold start latency and resource isolation and efficiency. Next, we investigate the suitability of various interpreted languages for usage in such a platform, before we describe our proposed platform's architecture for the lightweight Duktape and the fast Google V8 JavaScript engine. Finally, we outline attestation of Lambdas by users of our platform.

5.2.1 Requirements of a Secure FaaS platform

In general, a Lambda platform should be able to run larger numbers of Lambda scripts in parallel and isolated from each other, also supporting auto-scale depending on the current load. Lambdas should quickly be spawned if not already running, in order

[1] `https://aws.amazon.com/de/blogs/compute/container-reuse-in-lambda/`, last accessed 09/2020.

to achieve low response times for Lambda requests. Also, available system resources should be used efficiently, therefore a low memory footprint of each single Lambda and its context is favourable, in order to be able to hold many Lambdas in memory before being forced to evict them. In addition, execution of many Lambdas in parallel should be supported not only to fully use multi-core CPUs but also smoothen IO delays.

For a Lambda platform to become trustworthy, a few more aspects need to be considered. In order to support trusted Lambdas a trusted execution environment must be integrated into the platform to allow execution of trusted code, but also that code's integrity has to be ensured similarly to the approaches in the previous chapters of this thesis. This implies a procedure for deployment and storage of Lambda code on the platform in order to be able to launch Lambdas whenever required by the usage pattern.

Next, the confidentiality and integrity of network communication must be ensured, otherwise there would be no point of using trusted execution on the server-side. As the cloud provider is not trusted and the data that is processed is considered sensitive, the Lambda's execution state and all processed data must be protected. Usage of trusted execution technology like SGX naturally leads to this, as the enclave memory is encrypted, and thus, not available to the cloud provider. But even in case of using other (weaker) trusted execution technology like ARM TrustZone, the data processed by a Lambda would not be (easily) accessible by the cloud provider, except for physical attacks such as cold-boot attacks.

In order to achieve good performance with a trusted Lambda platform as outlined above, a few new factors need to be incorporated which are specific to the trusted execution technology being used—SGX in case of our TSP. In general, the enclave size should be as small as possible in order to prevent SGX paging as long as possible, as this induces a high performance impact [10]. Furthermore, as Lambdas are usually written in interpreted languages and comprise of relatively little code compared to the required runtime, sharing the runtime between multiple Lambdas allows increasing memory usage efficiency. This leads to an architecture that executes multiple Lambdas in the same enclave, and thus, additional strong isolation between Lambdas is required in order to ensure they can not access each other's data and monopolise resources. This includes isolation of subsequent requests to the same Lambda as well.

5.2.2 Lambda Programming Languages

During the process of designing a trusted Lambda platform we considered several programming languages for Lambdas. Thereby, selection of the type of programming

language—interpreted versus native—has intense implications on the Lambda platform. For example, an interpreted language requires an interpreter inside the TEE while native code does not, which is why we considered native Lambdas even though most existing commercial FaaS offerings focus on interpreted Lambda code. However, native code is not as portable as interpreted code and is in most cases not trivially capable of running in a TEE without at least some porting effort.

In principal, Lambdas implemented in compiled languages (e.g., C/C++ and Rust) would be advantageous, as their resource footprint is relatively small compared to interpreted languages that do require a full interpreter in order to execute. However, Lambdas are traditionally written in interpreted languages and we would like to support execution of existing Lambdas with minimal or at least automatable effort on top of our platform. In addition, native Lambdas would have to be ported and recompiled to be executable inside a secure enclave, as no system calls are available there without further measures. Even though we assume that Lambdas do not establish their own socket connections or access the file system directly, even simple actions like requesting the current time (`gettimeofday()`) may require system calls. While this is not strictly speaking a native code problem but also affects interpreted languages that require the same functionality, supporting such a functionality in native Lambdas requires some porting effort. This problem also affects library dependencies of Lambda code, as those libraries would have to be ported as well. Only approaches like Haven [15], Graphene [115] and SCONE [10] do allow execution of unchanged native applications inside secure enclaves, at the cost of a large runtime for example comprising a library OS inside the enclave. Also, isolation of native code is hard, as arbitrary memory locations can be accessed if no complex sandboxing (c.f Ryoan [52]) is implemented inside the secure enclave as well. If no native code sandboxing is feasible inside the enclave, the only remaining approach to achieve isolation is to execute native Lambdas each in their own enclave. However, this prevents any sharing of code such as the standard C library (libc) for example as shared memory between enclaves is not foreseen.

Code components written in interpreted languages could run without further changes inside a secure enclave, as long as the interpreter is available there. In addition to that, if the interpreted code has no library dependencies with native code parts, pure interpreted execution can be isolated from code in the same address space relatively easily, as memory access is controlled by the runtime. However, interpreted code is usually much slower than native code, especially if not accelerated by Just-In-Time (JIT) compilation.

Due to the above advantages of interpreted languages, their popular usage in FaaS systems, and the security issues of native code, we investigated several interpreted language environments and their suitability for usage in a secure FaaS scenario. Even

though most runtimes for interpreted languages offer at least some notion of a "context" to isolate code and share the runtime with others, they heavily rely on libraries and most of those libraries contain large fractions of native code components. This applies for example to Lua and Python. However, JavaScript has been originating from web browsers initially, that by design required the code to be platform independent and more or less self-contained. Therefore, many available JavaScript libraries are written in pure JavaScript code without any native dependencies. In addition, many serverless applications are written in JavaScript as JavaScript is a trending and popular language and all existing FaaS platforms (e.g. OpenWhisk[2]) support at least JavaScript amongst other languages [47]. For those reasons, we aim at supporting JavaScript-based Lambdas in our platform's architecture. However, we limit our platform to the execution of pure JavaScript code without any native components, due to the negative side effects of native code in shared enclaves as described above.

Nevertheless, even for JavaScript as a promising Lambda language candidate, there are still multiple options to execute JavaScript code. MuJS[3] was the first interpreter we looked into. While being extremely small and resource-efficient, MuJS has only quite limited ECMAScript 2015 support thereby inhibiting the use of modern JavaScript programming idioms. In contrast, the Duktape[4] JavaScript engine provides good—but still partial—support of ECMAScript 2015 and other features such as the ES2015 TypedArray and Node.js Buffer bindings, for example. Supporting ECMAScript 2015 would be beneficial, as regular JavaScript code can be automatically transpiled to ECMAScript 2015, but not necessarily to older versions as well. Finally, there is Google V8 as one of the most modern JavaScript engines with features for high performance such as JIT compilation and more sophisticated garbage collection approaches. While providing the most holistic language support, Google V8 with $\approx$ 1.3 million SLOC is by far also the largest of the engines we investigated in terms of its TCB and memory footprint. However, according to our measurements (*3dcube* and *base64* benchmark of the JetStream suite[5]) Google V8 performs about 30$\times$ to 84$\times$ better than Duktape.

With the properties of the individual approaches discussed above in mind, in the following we will aim at providing a platform that runs multiple *pure* JavaScript-based Lambdas on top of a single JavaScript engine inside the same enclave. With this approach we are able to share the relatively large interpreter between multiple Lambdas.

As it is unrealistic to assume Lambdas will get along without any dependencies, there

[2] `https://openwhisk.apache.org`
[3] MuJS JavaScript Engine `http://mujs.com/`
[4] Duktape JavaScript Engine `https://duktape.org/`
[5] JetStream JavaScript Benchmark Suite: `http://browserbench.org/JetStream/`

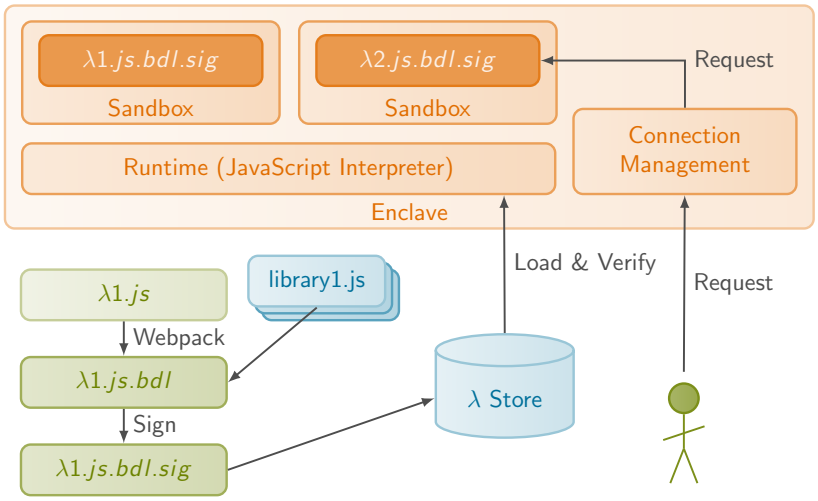


Figure 5.1: Trusted Serverless Platform Generic Architecture.

must be a way to support library dependencies of Lambdas running on our platform. Instead of loading libraries on demand from the outside, dependencies could also be bundled with the Lambda code into a standalone script. The advantage of this is, that the Lambda is represented by a single file with a signature inside the Lambda storage system, and can easily and quickly be loaded into the enclave with a single call. Also, in that case, the signature naturally includes the libraries used by the Lambda. In order to achieve this, we used *webpack*[6]. Webpack allows automatic resolving of calls from JavaScript code to the `require()` function, and downloads and bundles all required library dependencies recursively with the Lambda into a single standalone file.

5.2.3 TSP Architecture Overview

In principle the generic architecture of our TSP shown in Figure 5.1 is independent from the JavaScript interpreter being used. Still, the interpreter is the heart of the platform, running once inside the enclave and being shared between all Lambdas executed within that enclave. All Lambdas are executed in their own context, in order to be able to run in parallel and to provide reasonable isolation between them. This makes sense even if the same Lambda script is executed multiple times in different contexts in order to improve performance by parallel execution in high load situations.

In general, Lambdas are stored outside the enclave in the form of a signed (and optionally encrypted) bundle of the Lambda's JavaScript code on the file system. This bundle is being loaded on demand by the platform from the untrusted store into a

[6]webpack.js `https://webpack.js.org/`

newly created context and prepared for being called by users. During the loading process, the Lambda bundle's signature is being verified by the platform in order to ensure that only Lambdas correctly signed by valid customers are executed on the platform.

After a Lambda is loaded and verified a new context is created for its execution with the JavaScript interpreter. This ensures the Lambda is executed independently and isolated from other Lambdas and can run in parallel with other Lambdas. On high load on a single Lambda it may also be beneficial to instantiate multiple contexts for the same Lambda, in which case the Lambda is only loaded once from the outside, but multiple independent contexts are created from it.

A Lambda is loaded only on-demand, when a request for this Lambda arrives from a user and the Lambda is not yet present in the enclave. A new connection also requires a connection context to be created that stores connection-specific data such as the TLS session keys. In general, a connection context is independent from a Lambda context, as the connection context stores meta information about a connection to the platform by a user, while the Lambda context comprises data about that specific Lambda. Therefore, it is not required to link the two to each other, which also allows more flexibility such as reusing a connection context for a request by the same user to another Lambda in the same enclave for example.

5.2.4 Dynamic Load Adaptation

Instantiation of multiple contexts for one single Lambda allows exploiting multiple CPU cores in order to increase the overall system's performance (c.f. Section 5.3). However, the optimal number of contexts for one Lambda depends on the Lambda's code and usage pattern, and can not trivially determined statically or before Lambda deployment time. For this purpose, we developed a mechanism for dynamic adjustment of the number of Lambda contexts dependent from the current system load in order to optimise performance automatically.

In order to adjust the number of created Lambda contexts to the current load situation on the system, we introduce a μ-value, that describes the pressure on a specific Lambda context. The μ-value is calculated by the number of requests for each individual Lambda script in a given time frame, divided by the amount of "waits" that are required to find an available Lambda context and is stored by the platform for each Lambda script individually. When a request for a specific Lambda script is being processed, the platform will first try to find an unused context for this script and wait until a context is being released by another thread if none is available. If too many "waits" are required, the μ-value will decrease, and the platform can spawn new contexts for a

Lambda script once the μ-value falls below a configurable threshold. This ensures that the amount of contexts created for a Lambda script adjusts depending on the amount of computation and the number of requests per second in order to improve the performance and overall throughput of the platform.

The above μ-value adjusts the concurrent contexts of one Lambda on a single host. We assume that coarser-grained adjustment on a larger scale is additionally done by the untrusted cloud platform, balancing load across different machines and enabling scalability of Lambda applications. This can be done by the untrusted cloud provider using traditional and established technical means, and at the same time respecting properties like locality in approved placement policies.

5.2.5 TSP with the Duktape JavaScript Engine

At the heart of the TSP platform is the JavaScript interpreter. Following our goals of a lightweight platform due to the above discussed security and performance aspects, we investigated building our TSP with the lean *Duktape* JavaScript engine[7]. The TSP variant with the Duktape JavaScript engine is called *SeDuk* in the following and is one possible manifestation of our generic TSP platform architecture. The Duktape engine is embedded into an Intel SGX SDK enclave as a library, and is being used by our platform application to interpret the Lambdas that are loaded from the Lambda store. This instance of our architecture uses Duktape JavaScript contexts for Lambda isolation.

Requests issued by users are transmitted via a TCP socket opened by our application and are integrity- and confidentiality-protected during their transmission by TLS with the TLS endpoint *inside* the enclave.

The so called *LambdaManager* component inside the enclave is responsible for managing the life cycle of our Lambda contexts. When a context is required for a specific Lambda script, the script is only loaded from the outside Lambda store if not yet available inside the enclave. Contexts are also created on demand and reused for multiple invocations of the same Lambda. In case of very high load, multiple contexts for the same Lambda are instantiated by the platform guided by our μ-value.

5.2.6 TSP with the Google V8 JavaScript Engine

The above described TSP manifestation with the Duktape JavaScript engine pursues the idea of our TSP in its purest form. This section describes another manifestation of the TSP with the Google V8 JavaScript engine at its heart—we call this variant *SeGoo*. Its architecture differs in details from our generic one and is illustrated in Figure 5.2.

[7] Duktape JavaScript Engine `https://duktape.org/`

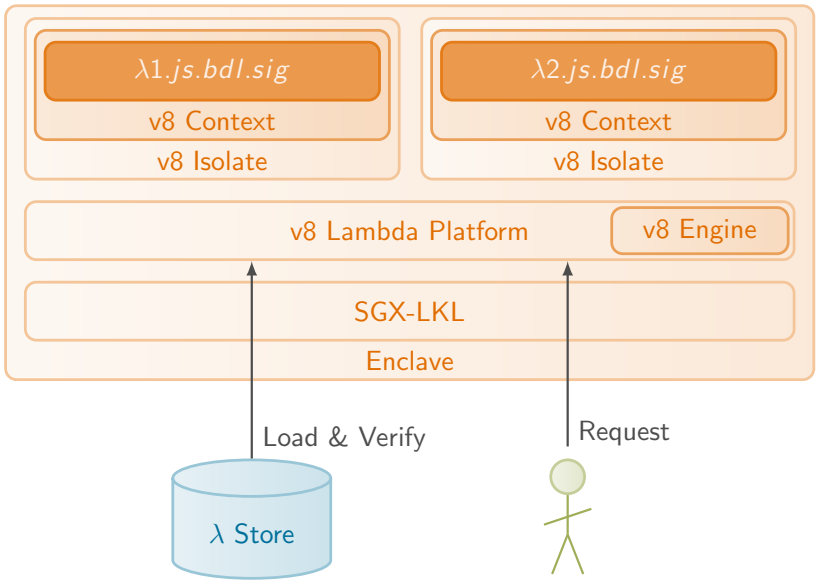


Figure 5.2: Trusted Serverless Platform Architecture with Google V8 on LKL.

In order to be able to execute Google V8 inside an enclave we used the SGX-LKL[8] project that combines the SGX technology and LKL, which is the Linux kernel as a linkable library. SGX-LKL resembles other approaches for execution of unchanged binary applications inside enclaves (e.g., by using a library-OS) such as [10, 115, 106, 15]. The enclave is not created by the Intel SGX SDK in this case, but instead uses a custom re-implementation of the SGX enclave creation and management process as part of SGX-LKL. With SGX-LKL there is user-level threading inside the enclave, as well as support for synchronisation and coordination of multi-threaded applications by using mutexes and conditional variables solely inside the enclave. In addition, SGX-LKL allows its guest application to issue system calls, that are processed asynchronously by threads outside the enclave. We built our *SeGoo* platform on top of SGX-LKL and linked it against the Google V8 JavaScript engine compiled for the *musl libc*[9] library, as this is required to run the application inside an SGX-LKL-based enclave.

Similarly to *SeDuk*, also *SeGoo* opens a TCP socket in order to listen to user requests. However, in this case the system calls related to the socket are handled by the asynchronous system call queuing mechanism of SGX-LKL and issued to the host kernel running outside the enclave. Still, confidentiality and integrity is protected by a TLS encryption that terminates inside the enclave.

The *SeGoo* application also maintains a *LambdaManager* component, just like *SeDuk*.

[8] SGX-LKL at Github: `https://github.com/lsds/sgx-lkl`
[9] musl libc `https://www.musl-libc.org/`

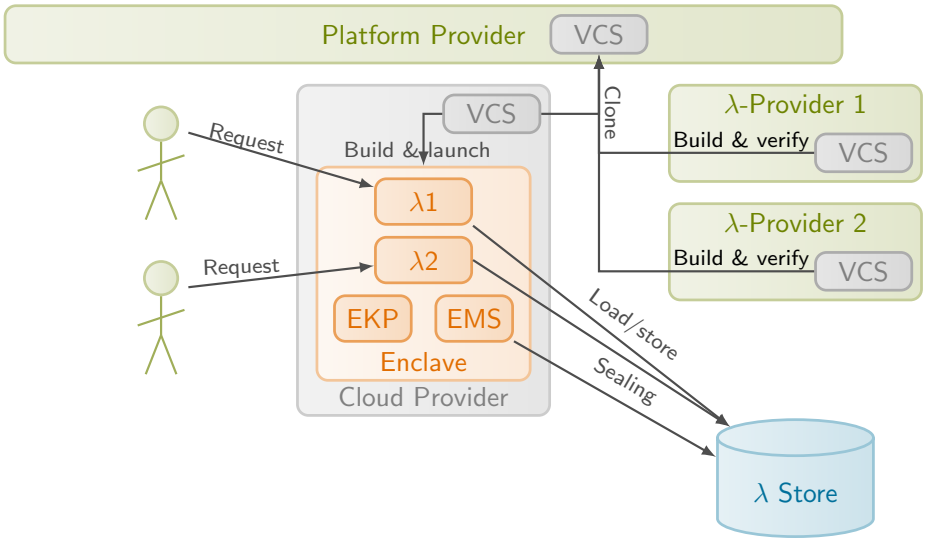


Figure 5.3: Trust and Key Management in Trusted Serverless Platform.

However, Lambdas are not only isolated by V8 contexts, but by V8 isolates that each comprise exactly one context in our case. This leads to a stronger isolation of Lambdas in contrast to *SeDuk*, as V8 isolates have been designed from the ground up with the idea of mutually distrusting scripts running in different web browser tabs in mind.

5.2.7 Trust and Key Management

In Figure 5.3 we describe key management and trust relationships of our platform. The figure introduces the following entities: the *cloud provider* owns the hardware and runs the enclave and the platform software (either *SeDuk* or *SeGoo*). The *Lambda providers* run their own Lambdas on top of the platform. The *platform provider* is the entity that develops and distributes the secure Lambda platform's software. Finally, the *users* issue requests and use the Lambdas via secure TLS connections terminating inside the enclave. Not all entities must be distinct, but may also be the same, for example, a Lambda provider can also be a Lambda user at the same time (e.g., in case of Lambda chaining).

Furthermore, we define two keys, the Enclave Master Secret (EMS) and the Enclave Key Pair (EKP): the EMS is a symmetric key used for confidentiality protection of data stored in the untrusted λ-store—a simple variant of a key-value store (KVS) hosted by the cloud provider. This EMS key is generated by the enclave and can be migrated from one enclave instance to another after successful mutual attestation, allowing the cloud provider to scale the platform to multiple instances. The EMS is stored itself inside the KVS (after sealing) to allow correct enclaves to automatically bootstrap the system. This

implies that for the trust management procedure described in this section to work, we assume availability of a sealing mechanism provided by the SGX platform as a basic building block. In addition, an enclave will generate (at enclave start) the EKP used for securing the connection between Lambda providers and the enclave[10]. For this to work, the Lambda providers each have to establish trust into the platform once via remote attestation, and retrieve that public key during the attestation process in order to protect all future communication with the platform using it.

In order to establish trust into the Lambda platform, the Lambda provider acquires the platform's source code (e.g. via a Version Control Systems (VCS)) and verifies and builds it in order to generate the expected hash value (`MRENCLAVE`) of the platform's binary. This of course assumes a deterministic build process and build environment in order to be able to produce an identical compilation binary which is already a complex task by itself. Next, the Lambda provider attests the platform running in the cloud by remote attestation with a nonce (for freshness) and the known hash value of the binary. The platform returns the public key of the EKP and an SGX attestation quote with a signature that can be verified using the IAS. This ensures authenticity of the EKP's public key, as the quote offers a user-data field that can be used to protect the identity of the EKP's public key (c.f. Section 2.4.4 for details about SGX attestation). After successful attestation, the Lambda provider can send sensitive data encrypted with the EKP public key to the platform. For example an encryption session key can be transmitted, which allows establishment of a secure connection between the enclave and the Lambda provider. In turn, this allows uploading Lambdas and TLS keys stored by the platform inside the untrusted KVS confidentiality-protected by the EMS.

The cloud provider initially also acquires the source code of the platform software from the platform provider and verifies it to ensure it will not harm her infrastructure. After a successful verification, the cloud provider builds and deploys the platform software and is henceforth only responsible for maintaining its availability.

After a Lambda is uploaded and stored inside the KVS along with at least one TLS key per Lambda provider, a Lambda is ready to process user requests. Requests are addressed to a specific subdomain identifying the Lambda provider, which allows the user to implicitly detect that the Lambda provider has successfully attested the platform, as only then a valid TLS connection for said domain is possible. In addition, this approach could even support multiple different Lambda providers in the same enclave, as long as each Lambda provider uses her own TLS key. By using Server Name Indication (SNI) the platform can maintain multiple TLS endpoints for distinct subdomains under the

[10] Our two prototypes use the Mbed TLS library (`https://tls.mbed.org/`)

same socket, all terminating inside the enclave.

This approach implies Lambda execution with transparent attestation by users solely through their requests over a valid TLS connection. Integrity of the Lambda code can be protected by an HMAC stored with the Lambda code inside the KVS. Note that during transmission the Lambda's code is already integrity-protected by TLS and inside the enclave by the SGX memory encryption.

In our concept the cloud provider is not to be trusted by any other entity. We allow multiple Lambda providers on the same platform with distinct keys, and enable users to only trust selected Lambda providers. Derived from the fundamental trust into the platform, we support Lambda-specific sealing of data (into the KVS) using a Lambda-specific key derived from the EMS and a hash value of the Lambda's code.

5.2.8 Aspects of Security

Lambdas must be detained from reaching outside their projected environment and harm the cloud provider or the platform. Furthermore, it is crucial to ensure that one Lambda can not access any data of another Lambda. For this reason, Lambdas are isolated from each other using container-based isolation mechanisms in many existing FaaS platforms—AWS and OpenLambda use Docker containers for example [47].

In addition, besides the high porting effort of native code to enclaves, one of the strongest arguments against native code even in the form of library dependencies of Lambdas is the required isolation of Lambdas. As native code components work with pointers, a large and complex sandbox or hardware mechanism must be brought in place to isolate them from each other [52]. By abandoning support for native components, isolation becomes much easier as almost all existing runtimes for interpreted languages already posses a notion of (isolated) contexts to run multiple scripts independently from each other. Even though previous executions of Lambdas will leave pre-owned objects behind, there is no way to access them from interpreted code before the garbage collector eradicates them due to the lack of valid references.

The problem with existing FaaS platforms in an SGX scenario is that the interpreter can not be shared between multiple Lambdas and must be instantiated for each Lambda. This leads to high memory consumption which is particularly difficult as SGX can only maintain good performance of the transparent memory encryption for small memory ranges (c.f. Section 2.4.2). The only option with SGX to achieve higher performance and more efficient resource usage is to co-locate multiple Lambdas inside one enclave using the same interpreter, however, isolation between Lambdas becomes essential in this case. For isolation, process-based isolation would be the best option, as this is con-

sidered a strong mitigation against a Speculative Side-Channel Attack (SSCA) [116], but this again would prevent sharing the interpreter between Lambdas.

We believe this dilemma could be relaxed by definition of policies negotiated by cloud customers and the provider that specify the required security-levels of Lambdas. For instance, for highly sensitive Lambdas the provider could be advised to start dedicated enclaves while other less sensitive Lambdas may be co-located with (a) Lambdas of the same provider or even (b) Lambdas of other Lambda providers. Obviously, with this approach highly sensitive Lambdas will lead to higher cost as they require more resources, but this allows the cloud provider to assign a price tag depending the required level of security. Enforcing those policies relies on an attested and trusted base platform inside the enclave which is independent from other Lambdas running in the same enclave, as described in Section 5.2.7.

Another relevant security property is performance isolation of Lambdas that ensures Lambdas can not stall execution and jeopardise liveness of the platform. This can be guaranteed by a small patch to the JavaScript interpreter, that calls `yield()` regularly. In case of *SeGoo* this triggers the internal scheduler of the user-level threading of SGX-LKL inside the enclave and allows a renegotiation of resource assignment. In case of *SeDuk* each request is handled by a distinct connection thread, therefore, liveness can be guaranteed by the untrusted cloud platform with established and mature procedures. Since a denial of service by the cloud provider can inherently not be prevented anyway, relying on the generally untrusted provider in regards of availability and liveness of the platform poses no additional risk.

For SGX-LKL-based Lambdas, memory usage can similarly be controlled by involving additional checks into the respective system calls for memory allocation in order to prevent Lambdas from allocating too much memory. In case of *SeDuk* an according check functionality can be integrated into the interpreter itself.

Direct access to persistent storage by Lambdas is not envisaged, instead Lambdas are supposed to access persistent storage via calls to the platform, similarly as Lambda input and output is implemented. Thereby the platform is supposed to restrict access to a reasonable and configurable amount of persistent storage.

5.3 Evaluation of the Trusted Serverless Platform (TSP)

In this section we evaluate the performance and security of the proposed trusted Lambda platform in its two peculiarities *SeDuk* and *SeGoo*. The overall security is dependent from the TCB and the enclave interface of the platform, while the performance is measured by the footprint of the memory working set and the level of parallelism. We

evaluate not only the request throughput of the platform but also performance criteria such as the cold start latency and response time of a Lambda, which is particularly relevant for such a system to compete.

Subject to our evaluation are several example Lambdas as use cases: a very simple "echo" Lambda, which returns the provided input to the callee, the "jpeg" Lambda, that processes a JPEG encoded image provided as Base64-encoded Lambda input and converts it to a bitmap image, as well as the "fibonacci" Lambda which calculates the 1250th Fibonacci number. In addition to that, we constructed two Lambdas by porting their respective JavaScript code from the official JetStream benchmark suite[11] in order to allow running them on our platform. Those two Lambdas are denoted as "base64" and "3dcube" in the following. While "base64" encodes random bytes into Base64, "3dcube" executes a 3D cube rotation benchmark.

In general, the use case Lambdas need to be adjusted to run on our platform first. For this purpose the code is encapsulated in a function body with a specific signature that is predefined by the Lambda platform. Even in case of the two Lambdas originating from the JetStream JavaScript benchmark, this "porting effort" is negligible. After porting the JavaScript code, the use case Lambdas can be deployed on the Lambda platform, i.e. they are stored in the platform's Lambda store, and are ready to be called.

Once a Lambda is called, the Lambda platform initialises the sandbox environment to run the Lambda within, and forwards the provided input of the call to the Lambda. It is now the Lambda's responsibility to process the input and provide the output value which is returned back to the caller by the platform. This flow is denoted as a *Lambda call* in the following of this thesis, and comprises the above described steps but explicitly excludes the deployment of a Lambda on the Lambda platform. The definition of this term is particularly relevant for the Lambda call measurements of Lambda running on our platform in Section 5.3.3.

5.3.1 Platform Security

In Table 5.2 we illustrate and compare the size of the code base of the individual components of our two platforms. In this table, "Interpreter" represents the JavaScript engine itself that is being used to interpret the Lambda's JavaScript code. "Environment" stands for the required Intel SGX SDK libraries in case of *SeDuk*, and the LKL in case of *SeGoo* which is required inside the enclave in order to run the application. Finally, "Platform" represents our platform application (*SeDuk* and *SeGoo* respectively) which bridges the gap between the environment and the engine and takes care of all man-

[11] JetStream JavaScript Benchmark Suite: `http://browserbench.org/JetStream/`

Table 5.2: Size of Code Base in Lines of Code.

	SeDuk	*SeGoo*
Interpreter	185,392	1,308,702
Environment	214,156	17,193,624
Platform	1,529	1,002
Sum	401,077	18,503,328

agement tasks like managing JavaScript contexts, loading Lambdas from the untrusted storage and managing their life cycle, as well as the client connection management.

As can be seen from the table, *SeDuk* is much smaller in terms of the required source code when compared to the *SeGoo* platform. This is mainly due to the much smaller (and slower) Duktape interpreter used in *SeDuk*, but also due to the LKL required to run Google V8 in *SeGoo* which is much larger than the Intel SGX SDK. The code base of the Duktape-based platform is about 46× smaller than the Google V8-based platform.

Another relevant security aspect is the interface of the enclave to the outside world—the *untrusted interface*. *SeDuk* is based on a tailored enclave and only offers a few relatively specific ecalls such as `init()`, `call()` or `listen()`. In addition to that, the mbedTLS-SGX library adds common ecalls derived from socket syscalls such as `connect()`, `bind()`, `accept()`, `recv()` and `send()`. In contrast, *SeGoo* is based on the LKL library OS which requires a larger number of more than 20 ecalls, comparable to other library OS-based systems [10]. Hence, *SeDuk* is considered more secure than *SeGoo* in this regards due to the more specific and leaner enclave interface to the outside environment.

5.3.2 Lambda Throughput

This section of the evaluation covers the measurements of the main performance metric, the Lambda request throughput. The experiments have been performed on SGX-capable hosts with an Intel Core i7-6700 CPU, 24 GB memory, 1 Gbps network interface and SGX with 128 MB EPC. For this measurement, we run V8 directly on the host (without usage of SGX) as the baseline for the experiment. This has been accomplished by usage of the Google V8-based platform's binary and its execution directly on the host inside an Alpine Linux Docker container in order to supply the required musl-libc library. Then, we compare the baseline against the two platform variants *SeDuk* and *SeGoo*. *SeDuk* represents the Intel SGX SDK-based application including the Duktape JavaScript engine inside the enclave, while *SeGoo* is the Google v8-based native musl-

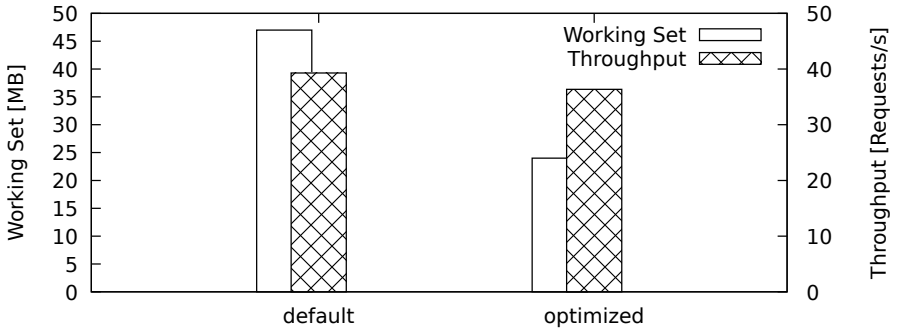


Figure 5.4: Effects of Google V8 active memory optimisation (`-optimize-for-size`).

libc application running inside an SGX enclave provided by SGX-LKL. The results of this benchmark are created using the *h2load*[12] HTTP benchmark with 8 parallel client threads and a separate warm-up phase to stabilise the results. For each of the three systems, we provide measurements for the five distinct Lambdas introduced above. Note that hereby "echo" effectively illustrates the platform overhead as this trivial Lambda solely consists of a JavaScript function returning its input.

As a preliminary experiment, we investigated the effects of the `-optimize-for-size` feature of Google V8 that optimises memory usage at the cost of execution speed, on the working set memory footprint and the throughput of our *SeGoo* application. The measurement results of the "base64" Lambda in Figure 5.4 clearly show, that the feature comes with an only slight performance degradation (7.4%) but quite significantly reduces the memory footprint of the application (51%), so it makes sense to activate this optimisation in most cases and it has been active for the remainder of this evaluation.

The results of another preliminary experiment are illustrated in Figure 5.5 which shows the overall throughput for an increasing (static) number of contexts for the *jpeg*, *base64* and *3dcube* Lambda on *SeGoo*. This experiment shows that the overall platform throughput can be increased by instantiating multiple Lambda contexts for the same Lambda in order to execute multiple requests to the same Lambda in parallel and incorporate the advantages of multi-core platforms. As can be seen, with increasing number of contexts the overall throughput increases until the system is saturated. However,

[12] h2load `https://nghttp2.org/documentation/h2load.1.html`

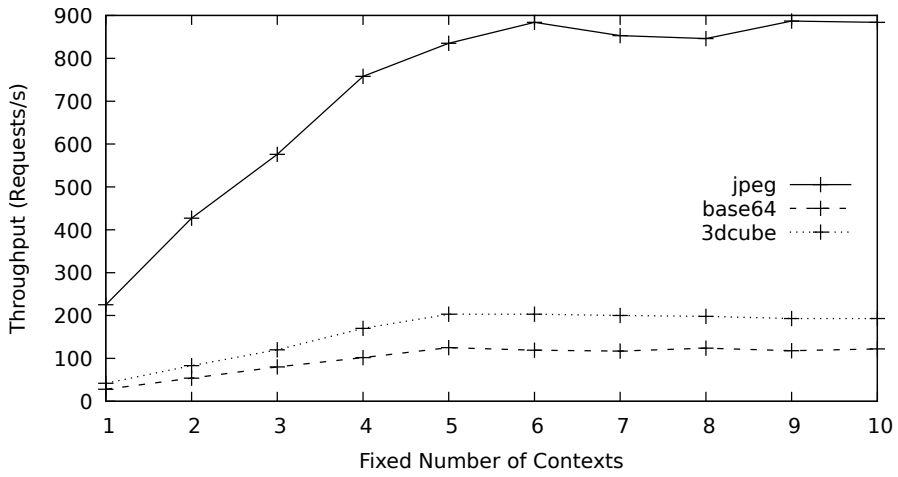


Figure 5.5: Fixed number of Lambda contexts for *jpeg*, *base64* and *3dcube*.

saturation is not always reached at the same fixed number of contexts, which motivated the introduction of our μ-value (c.f. Section 5.2.4) which is used in order to determine the point of saturation for each individual Lambda dynamically.

Finally, Figure 5.6 shows the results of the overall throughput measurements comparing the two secure Lambda platforms against the baseline. For this experiment we let both platforms decide for the optimal number of parallel contexts based on the μ-value (c.f. Section 5.2.4). At first glance, it can be seen that the throughput of both platforms correlates with the baseline. Also, the *SeDuk* platform is more lightweight and induces a lower platform overhead, as it includes the lean Duktape JavaScript engine. This can be seen in the measurement results of the *echo* Lambda where *SeDuk* achieves higher throughput as the baseline which, in contrast, implies usage of the more heavyweight Google V8 JavaScript engine. However, *SeGoo* performs significantly better than *SeDuk* for all other Lambdas, especially the ones with more complex computation (*jpeg*, *base64* and *3dcube*). As can be seen, the *SeGoo* platform is much faster than *SeDuk* which achieves approximately 6% of the performance of *SeGoo* for the complex Lambdas (*jpeg*, *base64*, *3dcube*) and 67.2% for the *fibonacci* Lambda. We explain this by the far more capable JavaScript engine which includes many optimisation features such as just in time compilation. However, in case of the *echo* Lambda, the *SeDuk* application is even 5$\times$ faster than the baseline, emphasising the benefits of the lean *SeDuk* platform.

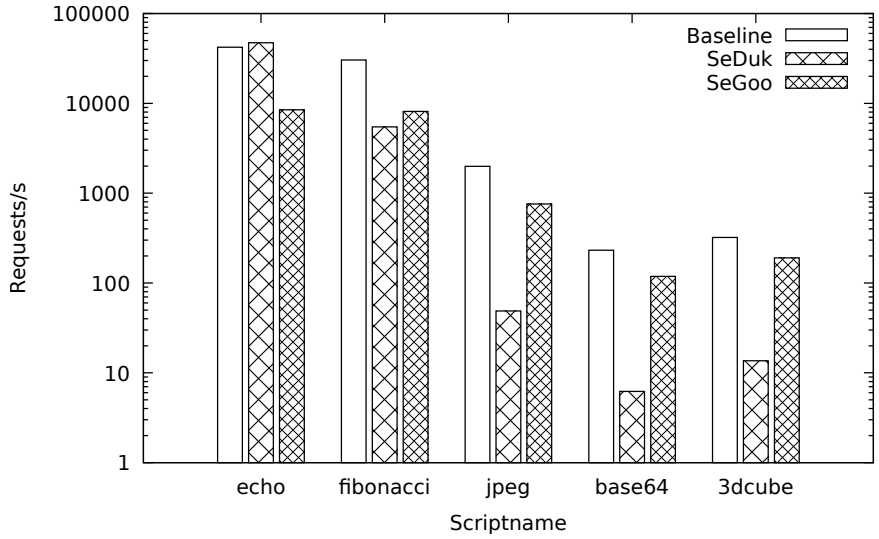


Figure 5.6: TSP Throughput Comparison.

5.3.3 Lambda Call Duration

We evaluated the Lambda call duration of our two platforms in various cases and show the results in Table 5.3. The measurements compare *SeGoo* and *SeDuk* against the same baseline as in the previous section. We compare the *warm* call duration where the Lambda script is already loaded into the enclave and a context is already created for it, with the *cold* call duration where the Lambda script has to be loaded from untrusted storage and a context must be first created. In the last row of the table, we also show the plain *overhead* of the platform including TLS encryption, enclave entering and exiting and context lookup, but *except* for the actual execution of the Lambda's JavaScript code. In addition, all measurements are done a) for a new connection, thereby including the TLS handshake, and b) with an already open connection to measure the overhead of creating a new connection. All measurements are average values of multiple test runs on the *3dcube* Lambda with an additional warm-up phase for more stable results.

As can be seen in Table 5.3, keeping a connection alive is beneficial, as well as reusing Lambda contexts. Also, while all platforms have a similar overhead, *SeDuk* has a much higher Lambda call duration, as Lambda processing by the Duktape JavaScript engine is slower. In most cases *SeGoo* has a slightly higher Lambda call duration than the baseline except for new connections on a warm Lambda. This is due to the asynchronous system call processing of SGX-LKL that even outperforms the application outside the enclave, as there is a high number of system calls during the TLS handshake. This also explains the high platform overhead for new connections on *SeDuk*.

Table 5.3: Lambda call duration (3D Cube Lambda).

	Baseline	***SeGoo***	***SeDuk***
	New connection		
Cold	120.7ms	144.7ms	265.8ms
Warm	101.0ms	94.0ms	265.5ms
Overhead	93.6ms	76.4ms	93.0ms
	Open connection		
Cold	46.4ms	82.2ms	172.3ms
Warm	16.7ms	18.1ms	170.9ms
Overhead	0.9ms	0.9ms	1.0ms

5.3.4 Memory Working Set

In order to measure the working set memory footprint of our *SeGoo* enclave, we used *sgx-perf* [120]: the tool expects the enclave start address and size as an input and derives the enclave memory range (`ELRANGE`). It then removes all page permissions from that range and registers a custom page fault handler. Once the enclave accesses a page, the custom page fault handler is notified and resets the page permission for that page. We can, at any time, reset the counters of *sgx-perf*, in order to exclude memory accesses only required during enclave initialisation, and thus, irrelevant for the runtime performance of the system. Hence, with the help of *sgx-perf*, we were able to measure the amount of pages that are actually in use during our benchmark after a warm up phase of 60 s which is responsible for the caches building up for example.

Figure 5.7 shows the memory footprint of our Lambdas running on *SeGoo* measured with the approach described above. The measurement proves the practicability and low memory footprint of the platform including the Lambdas, staying well below the SGX paging threshold of up to 128 MB, except for the `base64` Lambda which exceeds the threshold of 128 MB and induces SGX paging already with three simultaneous contexts. In order to compare the two described platforms in this specific metric, a working set memory footprint measurement has been executed for the *SeDuk* platform with the same methodology and under comparable parameters. This allows a rough estimate of how much memory *SeDuk* requires in comparison with *SeGoo*. It can be said, that the *SeDuk* platform with an approximately 38% smaller footprint requires a relatively high amount of memory compared with its significantly smaller code base.

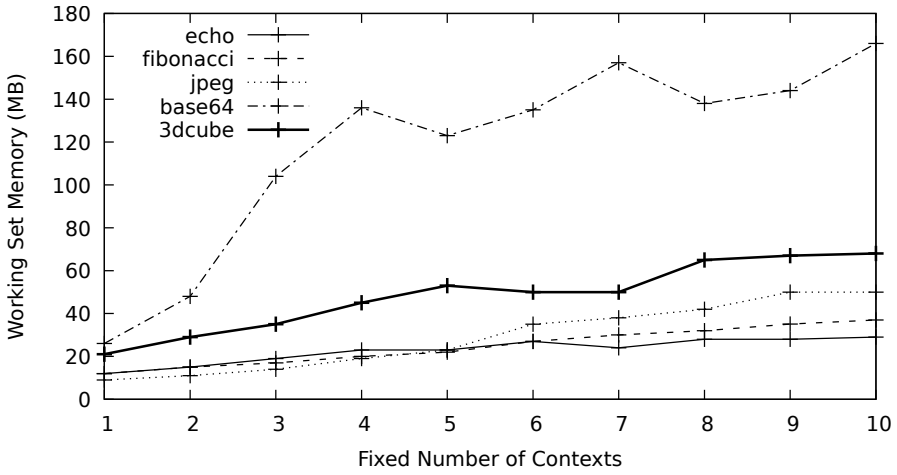


Figure 5.7: Enclave working set memory footprint.

5.4 Related Work

Several existing works have investigated the combination of trusted execution—especially SGX—with small actors or serverless-like functions.

Ryoan [52] allows users the processing of sensitive data in an untrusted cloud platform. The project builds upon Google's NaCl [123] sandboxing mechanism to protect the OS from the content of the user-defined containers. This approach would be required if Lambdas contained native components or dependencies. However, our TSP focuses on providing a lean trusted FaaS platform and the aspect of efficient resource utilisation as well as the execution of JavaScript-based Lambdas as the common denominator in this domain. In contrast to Ryoan we aimed at omitting the complexity of a native code sandbox like NaCl in the secure enclave which is central to their approach. In case of *SeGoo* our approach also comprises a relatively large code base that even includes the Google V8-integrated sandbox for WebAssembly which resembles NaCl. However, this large and complex component is unused in our case as we focus on JavaScript-only functions, and would be removed in a production environment to reduce the attack surface and improve security. In addition to that, Google V8 is not the preferred interpreter under our set goals, but instead only the better performing one in contrast to Duktape which fulfils our set goals of a lean platform much better.

Shen et al. [105] try to approach the inherent tension between isolation and sharing, especially in SGX applications, and propose a single-address-space solution comprising their library OS as well as all user-level applications in a single enclave. In their case,

isolation is achieved by leveraging the Intel MPX technology and the implementation of a Software Fault Isolation (SFI) scheme in order to support isolation between distrusting applications. In the scope of our TSP this approach is applied at compile-time and as mentioned requires an additional hardware capability, namely Intel MPX.

Boucher et al. [16] propose a FaaS architecture that focuses on small entities and language-based isolation, achieved by compiled micro services written in Rust. Their approach relies on compiled code and similarly as the work by Shen et al. [105] relies on compile-time guarantees for isolation and preemption. In contrast to their work, we focus on supporting Lambdas without requiring porting them for the platform.

Alder et al. [2] propose a trusted FaaS architecture based on the Duktape JavaScript interpreter. Their work is orthogonal to this thesis as they focus on the accounting of Lambdas in such a platform which is an aspect not covered in our efforts on the TSP.

Proceeding from the time of publishing TSP research in this field has continued and evolved. For example, Trach et al. [113] presented a FaaS platform that uses SCONE-based [10] SGX enclaves as a re-encryption proxy and function execution container. In contrast to our work, they integrated their work into the existing Apache OpenWhisk platform and support native and Python Lambdas running in SCONE containers. Wang et al. [118] have investigated running interpreters inside SGX enclaves as well. The authors enabled several interpreters to run inside of an SGX enclave atop and statically linked to a modified musl-libc library and allow for example Lua and JavaScript (based on the MuJS interpreter). In contrast, the authors of Diggi [41] investigated execution of native FaaS functions in SGX in order to reduce the TCB. However, their approach inherently can not support running unchanged existing Lambdas from widespread Lambda platforms and interpreted code in general, as opposed to our work.

5.5 Summary

In this section we have shown how serverless applications could be secured using trusted execution. For this purpose we designed a prototype architecture of a trusted serverless platform which we called TSP, and implemented two variants of it using two different JavaScript interpreters each with its own specific benefits and characteristics. With *SeDuk* we have shown, that a lightweight platform can be built upon the Duktape JavaScript interpreter, however, while featuring low general platform overhead this variant can only achieve limited overall performance. In contrast, *SeGoo*, our second variant of the TSP based on the Google V8 JavaScript interpreter is much more heavyweight, but it achieves a much higher performance due to complex additional interpreter features such as a JIT compiler.

From a security point of view, the *SeDuk* offers the leaner TCB while *SeGoo* is relatively heavyweight as in our specific case it required the SGX-LKL project with LKL inside the enclave. Still this approach is valid in practice as the trusted code base does not comprise any of the business secrets of the cloud provider which would be problematic as all trusted code needs to be published to allow (remote) attestation.

Isolation of untrusted code from distinct entities is an additional crucial factor for a platform like TSP as it is important to prevent one cloud customer to interfere with applications of another one. In this regards, we consider the *SeGoo* platform more secure as the isolation mechanism used in this platform variant (Google V8 isolates) has been built from the ground up with security in mind. In light of recent CPU security vulnerabilities (e.g. Meltdown [68], Spectre [60]), isolation using software mechanisms *within* one address space without hardware-based process isolation should be considered risky. This is relatively critical for the initial use case of the Google V8 engine in the Google Chrome browser, as the threat model here is to isolate multiple potentially malicious JavaScript scripts to protect the user. In case of TSP which supposedly runs in the cloud, special measures could be implemented to enforce Spectre mitigations in the CPU and protect end users from such attacks solely through security-aware Lambda providers. For example a responsible Lambda provider could detect whether or not a vulnerable hardware platform has these mitigations in place (with a cryptographically protected proof during remote attestation) and refuse deployment of Lambdas on an unpatched hardware platform otherwise.

6 Conclusion

This thesis targeted achieving trust into a generally untrusted cloud. The main overall challenge thereby lies in managing the balancing act of using computing resources of an untrusted party (the cloud provider) while processing sensitive data with those resources without the cloud provider being able to access that data.

6.1 Research Challenges

The first building block for achieving this was enabling trusted execution in an untrusted cloud setting in general. Therefore, we firstly investigated the ARM TrustZone technology as an example of a commonly available trusted execution technology in this thesis. In this context we have discussed the general issues and manyfold risks with trust in a cloud computing scenario as well as defined the requirements for achieving trust in such a setting. As a proof of concept and the first major contribution of this thesis, we designed and implemented the TrApps platform which allows execution of multiple trusted components isolated from each other in an untrusted cloud.

Next, in this thesis we examined the question how existing cloud-related applications could be secured using trusted execution technology, either on top of our ARM TrustZone-based TrApps platform or using the at that time newly available Intel SGX technology for x86 architectures. Hereby, we investigated the porting process of existing applications for trusted execution in an untrusted cloud environment by usage of trusted execution technology. In order to show the feasibility of this, the following contributions as part of this thesis were made: on the TrApps platform we designed and implemented the Secure Memcached application, while we presented the SecureKeeper and Dumbledore prototypes secured using the Intel SGX technology.

Finally, with serverless cloud computing we also investigated a modern software architecture and cloud computing paradigm. Hereby, we investigated how that newly popular paradigm of cloud computing could be secured using trusted execution and provided a generic Lambda platform that supports SGX-secured Lambda execution as the last major contribution of this thesis.

6.2 Enhancing Cloud Security with Trusted Execution

From the various presented prototypes we can conclude that integrity is an important basic requirement to build upon in order to create any kind of trust into a system remotely. Without protecting the integrity of a software component including the underlying hardware platform by means of remote attestation or a trusted boot, it could not be plausibly assured to a user that the intended and probably verified software stack is actually deployed and trust could not be established as a basic building block of sensitive data processing. Building on top of integrity, the confidentiality of application code is only optional while being desirable in certain cases, as scenarios are imaginable that can tolerate inspection of the code by the cloud provider as long as the secrecy of keys and user data is guaranteed.

The above hypothesis also implies that at least a very basic secret key storage is another crucial requirement for all trusted cloud platforms as without it no sensitive data processing is possible. As long as the cloud provides the means to store at least one key in a secret way, a secure vault containing more secret keys can be bootstrapped and secured with that single master key. Furthermore, in each trusted platform there must always be one key that initially establishes the root of trust where all trust is eventually derived from. In case of our ARM TrustZone-based TrApps platform there is the key that verifies the very first image booted on the hardware platform, while in case of SGX-based applications there is the key integrated into the SGX CPU during manufacturing. The protection of this root key is crucial as all trust is eventually derived from it.

Once a basic level of trust into a platform in the cloud is established and the requirements like a secret key storage are met for running a secure application, the next challenge is the development of such applications running with the help of trusted execution in an untrusted cloud. In this regard we have shown that partitioning of applications is feasible, but requires manual work for splitting the code base and identifying all sensitive code and data of an application. While approaches exist that try to automate that process, complete automation of this procedure has not yet been achieved.

Another approach, avoiding the partitioning of existing applications is the execution of complete applications unchanged on top of a trusted cloud platform. While this is arguably not feasible or at least not performing well for very large applications, we have shown for smaller components that running them unchanged without porting is possible with Lambdas on our secure serverless cloud platform. While it surely implies noticeable performance penalties, this still shows that a transparent security layer is feasible and an application could—under several assumptions as described in Chapter 5—be secured by flicking a switch (and paying a lot more). As computing resources

in a cloud setting are only a matter of money, this makes security mostly a matter of how much money one is willing to pay instead of an unsolvable problem or a problem that requires significant development overhead.

6.3 Outlook

After the above discussion of this thesis' contributions, this section covers future research directions building on top of the results of this thesis. As shown in Chapter 4, partitioning an application can be quite complex and cumbersome. Therefore, research in the direction of supporting that partitioning or porting process, right up to at least partially automating it, would be interesting. This task has already been initially investigated by Glamdring [67] for example.

Another interesting path would be to investigate more programming languages for our serverless platform, especially the popular Python language, in order to cover the two most popular languages for Lambda development. However, this requires efficient isolation of Python scripts running on the same interpreter in order to save resources, which is particularly difficult for Python libraries with native components.

Finally, in the future it is to be expected that the trusted execution technologies will improve and new ones will arise. In case of ARM, high performance TrustZone-capable hardware platforms are improving and the availability of hardware virtualisation inside normal world offers more opportunities especially for multi-tenant cloud platforms. In the scope of SGX it is to be expected that the limited amount of EPC memory will increase or the limit might eventually fall altogether. This would allow for relaxation of the relatively tight memory boundaries dealt with in this thesis. Finally, it is also relatively likely that other manufacturers might come up with similar technologies for hardware-assisted security measures.

Bibliography

[1] Divyakant Agrawal et al. “Managing Geo-replicated Data in Multi-datacenters”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2013.

[2] Fritz Alder et al. “S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX”. In: *ACM Cloud Computing Security Workshop*. 2019.

[3] Amazon. *Amazon Web Services: Overview of Security Processes*. 2014. URL: http://aws.amazon.com/security.

[4] *Amazon AWS Lambda*. https://aws.amazon.com/lambda/.

[5] *An update on 3rd Party Attestation*. https://software.intel.com/en-us/blogs/2018/12/09/an-update-on-3rd-party-attestation. 2019.

[6] Ittai Anati et al. “Innovative Technology for CPU Based Attestation and Sealing”. In: *International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013.

[7] *Apache OpenWhisk*. https://openwhisk.apache.org/.

[8] Masoud Saeida Ardekani and Douglas B. Terry. “A Self-Configurable Geo-Replicated Cloud Storage System”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[9] ARM. *Building a Secure System using TrustZone Technology*. Tech. rep. 2009.

[10] Sergei Arnautov et al. “SCONE: Secure linux containers with Intel SGX”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[11] Ahmed M. Azab et al. “Hypervision Across Worlds”. In: *ACM Conference on Computer and Communications Security*. 2014.

[12] Sumeet Bajaj and Radu Sion. “TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality”. In: *IEEE Transactions on Knowledge and Data Engineering* (2014).

[13] Ioana Baldini et al. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. 2017.

[14] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM Symposium on Operating Systems Principles* (2003).

[15] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding applications from an untrusted cloud with Haven”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[16] Sol Boucher et al. “Putting the Micro Back in Microservice”. In: *USENIX Annual Technical Conference (USENIX ATC ’18)*. 2018.

[17] Marcus Brandenburger et al. “Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks*. 2017.

[18] Ferdinand Brasser et al. “SANCTUARY: ARMing TrustZone with User-space Enclaves.” In: *Network and Distributed System Security Symposium*. 2019.

[19] Stefan Brenner, Michael Behlendorf, and Rüdiger Kapitza. “Trusted Execution, and the Impact of Security on Performance”. In: *Workshop on System Software for Trusted Execution*. 2018.

[20] Stefan Brenner, David Goltzsche, and Rüdiger Kapitza. “TrApps: Secure compartments in the evil cloud”. In: *International Workshop on Security and Dependability of Multi-Domain Infrastructures*. 2017.

[21] Stefan Brenner, Colin Wulf, and Rüdiger Kapitza. “Running ZooKeeper coordination services in untrusted clouds”. In: *Workshop on Hot Topics in System Dependability*. 2014.

[22] Stefan Brenner et al. “SecureKeeper: Confidential ZooKeeper using Intel SGX”. In: *ACM/IFIP International Middleware Conference*. 2016.

[23] Tiago Brito, Nuno O. Duarte, and Nuno Santos. “ARM TrustZone for Secure Image Processing on the Cloud”. In: *IEEE Symposium on Reliable Distributed Systems Workshops ARM*. 2016.

[24] Jo Van Bulck et al. “A Tale of Two Worlds: Assessing the Vul-nerability of Enclave Shielding Runtimes”. In: *ACM Conference on Computer and Communications Security*. 2019.

[25] Rajkumar Buyya et al. “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation Computer Systems* 25 (2009).

[26] Xiaoxin Chen et al. “Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems”. In: *Architectural Support for Programming Languages and Operating Systems*. 2008.

[27] *Cloudflare Workers*. `https://www.cloudflare.com/products/cloudflare-workers/`.

[28] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive* 2016 (2016).

[29] John Criswell, Nathan Dautenhahn, and Vikram Adve. “Virtual Ghost: Protecting Applications from Hostile Operating Systems John”. In: *Architectural Support for Programming Languages and Operating Systems*. Vol. 49. 2014.

[30] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2004.

[31] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *ACM Symposium on Operating Systems Principles*. 2007.

[32] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. 2017.

[33] Salessawi Ferede et al. “Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors”. In: *IEEE International Symposium on High Performance Computer Architecture*. 2017.

[34] Andrew Ferraiuolo et al. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *ACM Symposium on Operating Systems Principles*. 2017.

[35] Phani Kishore Gadepalli et al. “Challenges and Opportunities for Efficient Serverless Computing at the Edge”. In: *Symposium on Reliable Distributed Systems*. 2019.

[36] Tal Garfinkel et al. “Terra: A virtual machine-based platform for trusted computing”. In: *ACM Symposium on Operating Systems Principles*. 2003.

[37] *Genode Operating System Framework*. `https://genode.org/`. 2019.

[38] Craig Gentry. “Computing arbitrary functions of encrypted data”. In: *Communications of the ACM* 53 (2010).

[39] Joel Gibson et al. “Benefits and challenges of three cloud computing service models”. In: *International Conference on Computational Aspects of Social Networks*. 2012.

[40] Anders T. Gjerdrum et al. “Performance of trusted computing in cloud infrastructures with Intel SGX”. In: *International Conference on Cloud Computing and Services Science*. 2017.

[41] Anders Tungeland Gjerdrum et al. “Diggi: A secure framework for hosting native cloud functions with minimal trust”. In: *IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*. 2019.

[42] *Google Cloud Functions*. `https://cloud.google.com/functions/`.

[43] Michael Gruhn and Tilo Muller. “On the Practicability of Cold Boot Attacks”. In: *International Conference on Availability, Reliability and Security*. 2013.

[44] Le Guan et al. “TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone”. In: *International Conference on Mobile Systems, Applications, and Services*. 2017.

[45] Shay Gueron. “A Memory Encryption Engine Suitable for General Purpose Processors”. In: *IACR Cryptology ePrint Archive* (2016).

[46] Keiko Hashizume et al. “An analysis of security issues for cloud computing”. In: *Journal of Internet Services and Applications* 4 (2013).

[47] Scott Hendrickson et al. “Serverless Computation with OpenLambda”. In: *USENIX Workshop on Hot Topics in Cloud Computing* (2016).

[48] Owen S. Hofmann et al. “InkTag: Secure Applications on an Untrusted Operating System.” In: *Architectural Support for Programming Languages and Operating Systems*. 2013.

[49] Sanghyun Hong et al. “Go Serverless: Securing Cloud via Serverless Design Patterns”. In: *USENIX Workshop on Hot Topics in Cloud Computing*. 2018.

[50] Zhichao Hua et al. “vTZ: Virtualizing ARM TrustZone”. In: *USENIX Security Symposium*. 2017.

[51] Patrick Hunt et al. “ZooKeeper: wait-free coordination for internet-scale systems”. In: *USENIX Annual Technical Conference*. 2010.

[52] Tyler Hunt et al. “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[53] *Intel® Software Guard Extensions (Intel® SGX) SDK*. `https://software.intel.com/en-us/sgx-sdk`. 2019.

[54] Jinsoo Jang et al. “PrivateZone: Providing a Private Execution Environment using ARM TrustZone”. In: *IEEE Transactions on Dependable and Secure Computing* (2016).

[55] K. R. Jayaram et al. “Trustworthy Geographically Fenced Hybrid Clouds”. In: *ACM/IFIP International Middleware Conference*. 2014.

[56] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks*. 2011.

[57] Rüdiger Kapitza et al. “CheapBFT: Resource-efficient Byzantine Fault Tolerance”. In: *EuroSys Conference*. 2012.

[58] Taehoon Kim et al. “ShieldStore: Shielded in-memory key-value storage with SGX”. In: *EuroSys Conference*. 2019.

[59] Avi Kivity et al. “KVM: the Linux Virtual Machine Monitor”. In: *Proceedings of the Linux Symposium*. Vol. One. 2007.

[60] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *IEEE Symposium on Security and Privacy*. 2019.

[61] Kari Kostiainen et al. “On-board credentials with open provisioning”. In: *International Symposium on Information, Computer, and Communications Security*. 2009.

[62] Ambuj Kumar et al. *Self-Defending Key Management Service with Intel® Software Guard Extensions*. Tech. rep. 2018.

[63] Anil Kurmus et al. “Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring.” In: *Network and Distributed System Security Symposium*. 2013.

[64] Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. “Can homomorphic encryption be practical?” In: *ACM Conference on Computer and Communications Security*. 2011.

[65] Seung-seob Lee et al. “Smart and Secure: Preserving Privacy in Untrusted Home Routers”. In: *ACM SIGOPS Asia-Pacific Workshop on Systems*. 2016.

[66] Yanlin Li et al. “MiniBox : A Two-Way Sandbox for x86 Native Code”. In: *USENIX Annual Technical Conference*. 2014.

[67] Joshua Lind et al. “Glamdring: Automatic Application Partitioning for Intel SGX”. In: *USENIX Annual Technical Conference*. 2017.

[68] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *USENIX Security Symposium*. 2018.

[69] Samsung Electronics Co. Ltd. *An Overview of Samsung KNOX*. Tech. rep. 2013.

[70] Jing Luo, Chunhua Jiang, and Xia Yang. "Design and implementation of security OS based on TrustZone". In: *IEEE International Conference on Electronic Measurement and Instruments*. Vol. 2. 2013.

[71] David Lyon. "Surveillance, Snowden, and Big Data: Capacities, consequences, critique". In: *Big data & society* 1 (2014).

[72] Pieter Maene et al. "Hardware-based trusted computing architectures for isolation and attestation". In: *IEEE Transactions on Computers* (2018).

[73] Paulo Martins, Leonel Sousa, and Artur Mariano. "A survey on fully homomorphic encryption: An engineering perspective". In: *ACM Computing Surveys* (2017).

[74] Sinisa Matetic et al. "ROTE: Rollback Protection for Trusted Execution". In: *USENIX Security Symposium*. 2017.

[75] Jonathan M. McCune et al. "Flicker: an execution infrastructure for TCB minimization". In: *EuroSys Conference*. 2008.

[76] Jonathan M. McCune et al. "How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution". In: *Architectural Support for Programming Languages and Operating Systems*. 2008.

[77] Jonathan M. McCune et al. "TrustVisor: Efficient TCB reduction and attestation". In: *IEEE Symposium on Security and Privacy*. 2010.

[78] Frank McKeen et al. "Innovative instructions and software model for isolated execution". In: *International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013.

[79] Peter Mell and Tim Grance. *The NIST definition of cloud computing*. 2011. URL: `http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf`.

[80] *Microsoft Azure Functions*. `https://azure.microsoft.com/en-us/services/functions/`.

[81] Richard Ta-Min, Lionel Litty, and David Lie. "Splitting interfaces: Making trust between applications and operating systems configurable". In: *USENIX Symposium on Operating Systems Design and Implementation*. 2006.

[82] Amin Mosayyebzadeh et al. "A Secure Cloud with Minimal Provider Trust". In: *USENIX Workshop on Hot Topics in Cloud Computing*. 2018.

[83] Daniel Nurmi et al. "The Eucalyptus open-source cloud-computing system". In: *IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2009.

[84] *OpenFaas*. `https://www.openfaas.com`.

[85] Meni Orenbach et al. “Eleos: ExitLess OS Services for SGX Enclaves”. In: *EuroSys Conference*. 2017.

[86] Napoleon C. Paxton. “Cloud Security: A Review of Current Issues and Proposed Solutions”. In: *IEEE International Conference on Collaboration and Internet Computing*. 2016.

[87] Siani Pearson and Azzedine Benameur. “Privacy, Security and Trust Issues Arising from Cloud Computing”. In: *IEEE International Conference on Cloud Computing Technology and Science*. 2010.

[88] Marcus Peinado et al. “NGSCB: A trusted open system”. In: *Australasian Conference on Information Security and Privacy*. 2004.

[89] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Computing Surveys* (2019).

[90] Rafael Pires et al. “Secure Content-Based Routing Using Intel Software Guard Extensions”. In: *ACM/IFIP International Middleware Conference*. 2016.

[91] Raluca Ada Popa et al. “Building web applications on top of encrypted data using Mylar”. In: *USENIX Symposium on Networked Systems Design and Implementation*. 2014.

[92] Raluca Ada Popa et al. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. In: *ACM Symposium on Operating Systems Principles*. 2011.

[93] Gerald J. Popek and Robert P. Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17 (1974).

[94] Christian Priebe, Kapil Vaswani, and Manuel Costa. “EnclaveDB: A Secure Database using SGX”. In: *IEEE Symposium on Security and Privacy*. 2018.

[95] Arthur Rahumed et al. “A secure cloud backup system with assured deletion and version control”. In: *International Conference on Parallel Processing Workshops*. 2011.

[96] Konstantin Rubinov et al. “Automated partitioning of android applications for trusted execution environments”. In: *IEEE/ACM International Conference on Software Engineering*. 2016.

[97] Nuno Santos et al. “Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services”. In: *USENIX Security Symposium*. 2012.

[98] Nuno Santos et al. “Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones”. In: *Workshop on mobile computing systems and applications*. 2011.

[99] Nuno Santos et al. “Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications”. In: *Architectural Support for Programming Languages and Operating Systems*. 2014.

[100] Vasily A. Sartakov et al. “STANlite –a database engine for secure data processing at rack-scale level”. In: *IEEE International Conference on Cloud Engineering*. 2018.

[101] Vinnie Scarlata et al. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation*. Tech. rep. 2018.

[102] Felix Schuster et al. “VC3: Trustworthy data analytics in the cloud using SGX”. In: *IEEE Symposium on Security and Privacy*. 2015.

[103] *Secure Boot on i.MX50, i.MX53, i.MX 6 and i.MX7 Series using HABv4*. `https://www.nxp.com/docs/en/application-note/AN4581.pdf`. 2018.

[104] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. “OpenStack: toward an open-source solution for cloud computing”. In: *International Journal of Computer Applications* 55 (2012).

[105] Youren Shen et al. “To Isolate, or to Share?: That is a Question for Intel SGX”. In: *Asia-Pacific Workshop on Systems*. 2018.

[106] Shweta Shinde et al. “PANOPLY: Low-TCB Linux Applications with SGX Enclaves”. In: *Network and Distributed System Security Symposium*. 2017.

[107] Lenin Singaravelu et al. “Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies”. In: *EuroSys Conference*. 2006.

[108] Sal Stolfo and Steven M. Bellovin. “Measuring security”. In: *IEEE Security & Privacy* (2011).

[109] Raoul Strackx and Frank Piessens. “Fides: Selectively Hardening Software Application Components against Kernel-level or Process-level Malware Raoul”. In: *ACM Conference on Computer and Communications Security*. 2012.

[110] GlobalPlatform Device Technology. *TEE Client API Specification Version 1.0*. Tech. rep. 2010.

[111] Hongliang Tian et al. “SGXKernel”. In: *ACM International Conference on Computing Frontiers (CF)*. 2017.

[112] Hongliang Tian et al. “Switchless Calls Made Practical in Intel SGX”. In: *Workshop on System Software for Trusted Execution*. 2018.

[113] Bohdan Trach et al. “Clemmys: Towards Secure Remote Execution in FaaS”. In: *ACM International Systems and Storage Conference*. 2019.

[114] Trusted Computing Group. *Trusted Platform Module Library, Part 1: Architecture*. 2014.

[115] Chia-Che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *USENIX Annual Technical Conference*. 2017.

[116] *v8 dev: Untrusted code mitigations*. `https://v8.dev/docs/untrusted-code-mitigations`. 2018.

[117] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

[118] Huibo Wang et al. "Running Language Interpreters Inside SGX". In: *ACM Asia Conference on Computer and Communications Security*. 2019.

[119] Liang Wang et al. "Peeking Behind the Curtains of Serverless Platforms". In: *USENIX Annual Technical Conference*. 2018.

[120] Nico Weichbrodt, Pierre Louis Aublin, and Rüdiger Kapitza. "SGX-Perf: A performance analysis tool for intel SGX enclaves". In: *ACM/IFIP International Middleware Conference*. 2018.

[121] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining Lost Cycles with HotCalls". In: *Annual International Symposium on Computer Architecture*. 2017.

[122] Johannes Winter. "Trusted computing building blocks for embedded linux-based ARM trustzone platforms". In: *ACM workshop on Scalable trusted computing*. 2008.

[123] Bennet Yee et al. "Native Client: A sandbox for portable, untrusted x86 native code". In: *IEEE Symposium on Security and Privacy*. 2009.

[124] Fengzhe Zhang et al. "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization". In: *ACM Symposium on Operating Systems Principles*. 2011.

[125] Qi Zhang, Lu Cheng, and Raouf Boutaba. "Cloud computing: state-of-the-art and research challenges". In: *Journal of internet services and applications* 1 (2010).

[126] Yinqian Zhang et al. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *ACM Conference on Computer and Communications Security*. 2014.